DISTRIBUTED CONSTRAINT OPTIMIZATION FOR MULTIAGENT SYSTEMS

by

Pragnesh Jay Modi

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2003

Copyright 2003 Pragnesh Jay Modi

Acknowledgements

First and foremost, I thank my advisors, Wei-Min Shen and Milind Tambe. They are exemplary role-models, first-rate mentors and great friends. It is difficult to fully outline all the ways in which each has contributed to my growth as a scientist. It is impossible to fully express my gratitude.

Yan Jin, Victor Lesser, Paul Rosenbloom and Makoto Yokoo donated their valuable time to serve on my dissertation committee. Makoto Yokoo was both co-author and collaborator who played a key role in the development of this dissertation. Victor Lesser was a supportive voice who always knew what to say and when to say it. Paul Rosenbloom was a brilliant constructive critic. Yan Jin helped me see my work from an outsider's perspective. I sincerely thank them all for their time and effort.

I have been fortunate to have had great colleagues over the years who have assisted me in various ways. Paul Scerri, David Pynadath, Hyuckchul Jung, Ranjit Nair and the rest of the TEAMCORE research group were all supportive friends. The members of the Intelligent Systems Division of ISI and AI-Grads are people I will never forget.

They are responsible for making my graduate school years a wonderful experience. ISI is a truly unique and rewarding place to work because of them.

Finally, I thank my parents. They taught me what real life is about and gave me the confidence to pursue this dream.

Contents

List Of Tables

List Of Figures

Abstract

To coordinate effectively, multiple agents must reason and communicate about the interactions between their individual local decisions. Distributed planning, distributed scheduling, distributed resource allocation and distributed task allocation are some examples of multiagent problems where such reasoning is required. In order to represent these types of automated reasoning problems, researchers in Multiagent Systems have proposed distributed constraints as a key paradigm. Previous research in Artificial Intelligence and Constraint Programming has shown that constraints are a convenient yet powerful way to represent automated reasoning problems.

This dissertation advances the state-of-the-art in Multiagent Systems and Constraint Programming through three key innovations. First, this dissertation introduces a novel algorithm for Distributed Constraint Optimization Problems (DCOP). DCOP significantly generalizes existing satisfaction-based constraint representations to allow optimization. We present *Adopt*, the first algorithm for DCOP that allows asynchronous concurrent execution and is guaranteed to terminate with the globally optimal solution. The key idea is to perform systematic distributed optimization based on conservative

solution quality estimates rather than exact knowledge of global solution quality. This method is empirically shown to yield orders of magnitude speedups over existing synchronous methods and is shown to be robust to lost messages.

Second, this dissertation introduces *bounded-error approximation* as a flexible method whereby agents can find global solutions that may not be optimal but are guaranteed to be within a given distance from optimal. This method is useful for time-limited domains because it decreases solution time and communication overhead. Bounded-error approximation is a significant departure from existing incomplete local methods, which rely exclusively on local information to obtain a decrease in solution time but at the cost of abandoning all theoretical guarantees on solution quality.

Third, this dissertation presents *generalized* mapping strategies that allow a significant class of distributed resource allocation problem to be automatically represented via distributed constraints. These mapping strategies are signficant because they illustrate the utility of the distributed constraint representation. These mapping strategies are useful because they provide multiagent researchers with a general, reusable methodology for understanding, representing and solving their own distributed resource allocation problems. Our theoretical results show the correctness of the mappings.

Chapter 1

Introduction

Designing loosely-coupled agents that coordinate effectively requires reasoning about interactions between individual agent decisions. Often it is necessary that such reasoning be done by the agents themselves in a collaborative but decentralized manner. Satellite constellations [3], disaster rescue [20], human/agent organizations [5], intelligent forces [4], distributed and reconfigurable robots [39] and sensor networks [44] are some examples of multiagent applications where distribution is inherent in the domain and one has little choice in redistributing or centralizing decision-making to make things easier.

In these distributed domains agents must reason about the interactions between their local decisions in order to obtain good global performance. For example, distributed robots must collaboratively construct the best joint plan that accomplishes system goals.

Or, constellations of satellites performing collaborative sensing missions must schedule their observations to maximize collection of scientific data. Or finally, distributed sensors must coordinate the allocation of their sensing resources in order to accurately track multiple targets. In these examples and others, the local decisions made by agents and the interactions between them can have significant implications on the performance of the multiagent system as a whole.

This dissertation proposes the Distributed Constraint Optimization Problem (DCOP) framework to model the interactions between local agent decisions and support efficient reasoning while maintaining the requirements of decentralization. Previous research in AI has shown that constraints are a convenient yet powerful way to represent reasoning problems [33]. DCOP significantly generalizes existing multiagent research that has focused primarily on satisfaction-based problems [50] which is inadequate for problems where solutions may have degrees of quality or cost. Other multiagent researchers have considered only informal representations of distributed constraint optimization [8] [32] [43].

This dissertation answers three key questions not currently addressed in existing research. First, a key outstanding challenge is how to efficiently coordinate distributed decision-making while providing theoretical guarantees on quality of global solution. While some distributed methods for DCOP do currently exist, these methods either a) provide quality guarantees only by requiring sequential execution [16], or b) allow concurrent execution but provide no kind of quality guarantee [51]. This is a problematic state of affairs because sequential execution can be prohibitively inefficient, while lack of theoretical guarantees prevents generalization across domains.

Second, how can agents find approximate (suboptimal) solutions when time is limited? Previous approaches have typically abandoned theoretical guarantees in order to reduce solution time. This is inadequate because, aside from experimental results in specific domains, such methods can provide no characterization of global solution quality. In addition, the lack of worst-case bounds on algorithm performance is problematic in domains where unpredicted boundry-cases may unexpectedly arise or costs of failure are very high, as in space missions [3] and humans/agent organizations [5].

Finally, given that constraints are a convenient, powerful, well-accepted representation, another key challenge involves designing general purpose techniques for representing real world problems in the distributed constraints paradigm. While there is much existing AI research on how to represent centralized problems using constraints, little guidance currently exists on how to represent *distributed* problem in the constraint reasoning framework. Multiagent designers faced with a new domain are required to invest a substantial degree of effort to correctly represent their problem. In addition, the modeling effort is often specific to the application preventing reuse across domains.

1.1 Objective and Approach

The principal question addressed in this thesis is:

> **How can distributed agents coordinate their local decision-making to provide theoretical guarantees on global solution quality under conditions of limited time, memory and communication?**

We make the following assumptions about communication:

- Agents can receive messages. Agents can send messages to other agents if they know their name.
- Messages are received in the order in which they are sent between any pair of agents.
- Messages are either dropped or correctly received (or equivalently, corrupted messages can be detected and dropped).

Our general approach stems from the basic premise that in a distributed multiagent system where time and communication is limited, it is difficult to obtain exact knowledge of global solution quality. Thus, the approach taken was to develop a method whereby distributed agents can perform systematic distributed search based on conservative solution quality *estimates* rather than exact knowledge of global solution quality. In this way, we allow agents to make the best decisions possible with currently available information. If more information is asynchronously received from others and if time permits, agents can revise their decisions as necessary. Because estimates are ensured to be conservative, agents are never led astray and guarantees on global solution quality are available.

We show that this method of distributed optimization has a number of benefits. First, it allows concurrent, asynchronous execution which yields significant orders of magnitude speedups while requiring only polynomial space at each agent. Second, since agents are not entirely dependent on global information to make local decisions, it is robust to message loss. Finally, this method can be used to find approximate (suboptimal) solutions while adhering to user-defined bounds on distance from optimal, yielding a practical technique for performing principled tradeoffs between time and solution quality.

1.2 Contributions

The thesis makes several contributions to Distributed Constraint Optimization for coordination in Multiagent Systems. These contributions are listed below.

- The Adopt algorithm, the first asynchronous complete algorithm for Distributed Constraint Optimization. In Adopt, agents perform systematic distributed search based on conservative solution quality estimates in the form of lower bounds. By exploiting the concurrency allowed by lower bound based search, Adopt is able to obtain significant orders of magnitude efficiency gains over sequential methods. (Chapter 3)

- While the Adopt algorithm presented in Chapter 3 assumes reliable communication, we demonstrate the proposed method is robust to message loss. (Chapter 4)

- Bounded error approximation for when time is limited. Bounded error approximation allows agents to find approximate (suboptimal) solutions while adhering to user-defined bounds on distance from optimal. This method yields a practical technique for performing principled tradeoffs between time and solution quality when time is limited. (Chapter 4)

- A detailed complexity analysis and identification of tractable subclasses of distributed resource allocation. This provides researchers with tools to understand the complexity of different types of problems and to understand the difficulty of their own resource allocation problems. (Chapter 5)

- General mapping strategies for representing distributed resource allocation problems via distributed constraints. We provide guidance on how to represent such problems in our framework by first proposing an abstract model of Distributed Resource Allocation and then converting the model into a distributed constraint representation using a generalized mapping. This mapping is reusable for any Distributed Resource Allocation Problem that conforms to the given model and we can show theoretically that the mapping preserves the problem. (Chapter 5)

In addition, these contributions taken together provide an end-to-end methodology, (problem complexity analysis, representation via distributed constraints, distributed solution method) for solving a significant class of distributed resource allocation problems.

We briefly outline the organization of this dissertation. Chapter 2 proposes the Distributed Constraint Optimization Problem (DCOP) framework. Chapter 3 presents the Adopt (Asynchronous Distributed Optimization) algorithm. Chapter 4 considers solutions to two practical issues that arise when solving DCOP in real-world multiagent domains: Limited time and unreliable communication. Chapter 5 proposes a general technique for modeling a key class of multiagent problems in which a set of agents are required to intelligently assign resources to a set of tasks. Chapter 6 reviews related work. Finally, Chapter 7 concludes by summarizing results and contributions while Chapter 8 identifies directions for future research.

Chapter 2

Distributed Constraint Optimization Problem

In this chapter, we introduce an example domain that motivates our work followed by a precise formalization of the problem.

2.1 Motivation

Our motivation is the design of collaborative agents that work together to accomplish global goals. While this problem arises in a wide variety of domains, one specific application we will use to illustrate our problem is a distributed sensor network domain [44] that has received significant attention by multiagent researchers in recent years [18] [36] [41] [45]. However, we emphasize that the techniques presented apply to a broad range of problems.

A sensor network consists of multiple stationary sensors and multiple moving targets to be tracked. The goal is to accurately track the targets by allocating sensors to targets subject to the following restrictions. Each sensor is *directional*, i.e., the sensor is equipped with three radar heads, each covering 120 degrees and only one radar head may be active at a time. Figure 2.1 (left) is a picture of a sensor with the three radar heads shown. An autonomous agent running on-board each sensor controls the direction of the sensor. Three different sensors must be allocated to a target for it to be tracked accurately and it is not possible for a sensor to track more than one target. Sensors have limited sensing range so they can only be allocated to nearby targets. Agents are able to send messages to each other using low-bandwidth radio-frequency communication. However, communication range is also limited, so agents may only communicate with nearby agents. Figure 2.1 (right) shows 9 sensors in a grid configuration and 4 targets with their associated weights describing target importance. Assuming each target can only be tracked by the four closest sensors, it is clear that only two of the four targets can be tracked. The agents must coordinate the allocation of sensors to targets with the goal of minimizing the sum of the weights of the ignored targets.

Designing software agents to effectively coordinate in this domain is challenging for a number of reasons. We outline four key issues of concern in this work.

- *Inherent distribution.* In inherently distributed domains, no single agent has global control or global view. In the domain described above, each agent must individually decide which target it should track based on local information and information received through messages from other agents. As another example in distributed robotics, each robot is autonomous and decides its own actions. Existing approaches to coordination that require a centralized decision maker are not feasible.

- *Limited communication.* Communication may be limited for a variety of reasons in different domains. In the sensor domain, communication is limited due to both range and bandwidth, so we cannot collect all information in a single location where global allocations could in principle be calculated. Also, radio-frequency communication is unreliable and messages may be lost. In other domains, privacy restrictions and prohibitive costs on translating information into an exchangable format impose limits on communication.

- *Limited time.* In the sensor domain, time for determining allocations is limited because targets are moving. For a given set of targets in particular locations, agents must be able to determine global allocations fast enough to be useful to track the targets before they have moved too far. In other domains, agents must act to perform joint tasks or execute joint plans within some given time window.

- *Theoretical guarantees.* In order to confidently deploy multiagent systems, theoretical guarantees on allocation quality are needed. Heuristic methods that may fail on boundry cases and provide no bounds on worst-case performance are not sufficient. In the sensor domain, algorithms must be able to guarantee that the

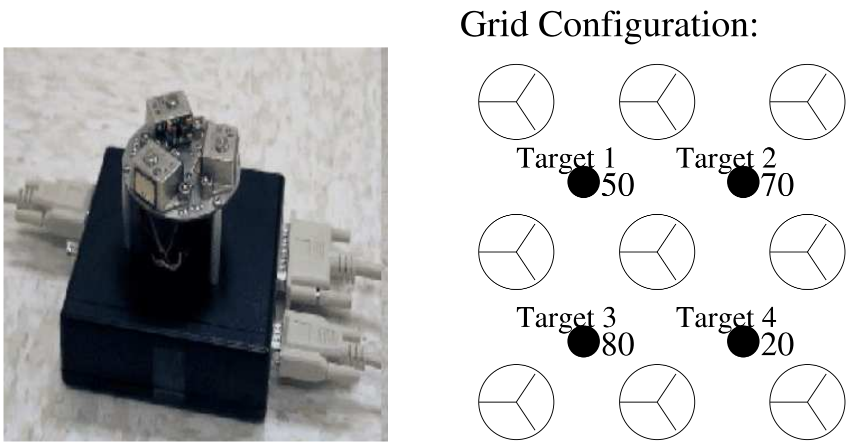


Figure 2.1: A hardware sensor (left) and schematic of nine sensors in a grid layout (right).

most important targets will get sufficient sensors allocated to them regardless of the particular situation. In multi-spacecraft missions proposed by researchers at NASA [3] [19], distributed satellites must collaborate to schedule/plan activities to obtain scientific data. Provable quality guarantees on system performance are required since mission failure can result in extraordinary monetary and scientific losses.

2.2 Background: Distributed Constraint Satisfaction

The Distributed Constraint Satisfaction (DisCSP) paradigm [50] has been proposed as a way to model and reason about the interactions between agents' local decisions. In DisCSP each agent controls the value of a variable and agents must coordinate their

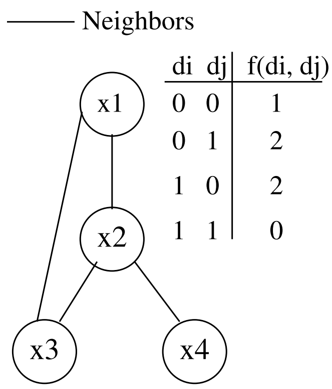


Figure 2.2: Example DCOP graph

choice of values so that a global objective function is satisfied. The global objective function is modeled as a set of constraints where each agent is only assumed to have knowledge of the constraints in which its variable is involved. Every constraint is required to be propositional, i.e., true or false. In this limited representation, an assignment of values to variables must satisfy *all* constraints in order to be considered a solution.

This representation is inadequate for many real-world problems where solutions may have degrees of quality or cost. For example, real-world problems are often *over-constrained* where it is impossible to satisfy all constraints. For these types of problems we may wish to obtain solutions that *minimize* the number of unsatisfied constraints. Next, we present a model that is able to capture this type of optimization problem.

2.3 Specification of DCOP Model

A Distributed Constraint Optimization Problem (DCOP) consists of n variables $V = \{x_1, x_2, ...x_n\}$, each assigned to an agent, where the values of the variables are taken from finite, discrete domains $D_1, D_2,..., D_n$, respectively. Only the agent who is assigned a variable has control of its value and knowledge of its domain. The goal is to choose values for variables such that a given objective function is minimized or maximized. The objective function is described as the sum over a set of cost functions, or valued constraints. The cost functions in DCOP are the analogue of constraints from DisCSP [50] (for convenience, we refer to cost functions as constraints). They take values of variables as input and, instead of returning "satisfied or unsatisfied", they return a valuation as a non-negative number. Thus, for each pair of variables x_i, x_j, we may be given a *cost function* $f_{ij} : D_i \times D_j \rightarrow N$.

Figure 2.2 shows an example constraint graph with four variables $V = \{x_1\ , x_2, x_3, x_4\}$. Each variable has two values in its domain, $D_i = 0, 1$. Assume each variable is assigned to a different agent. In the example, there are four constraints with the cost function as shown. All constraints have the same cost function in this example only for simplicity. In general, each constraint could be very different. Two agents x_i, x_j are *neighbors* if they have a constraint between them. x_1 and x_3 are neighbors because a constraint exists between them, but x_1 and x_4 are not neighbors because they have no constraint.

The objective is to find an assignment $\mathcal{A}^*$ of values to variables such that the total cost, denoted F, is minimized and every variable has a value. Stated formally, we wish to find $\mathcal{A}$ ($= \mathcal{A}^*$) such that $F(\mathcal{A})$ is minimized, where the objective function F is defined as

$$F(\mathcal{A}) = \sum_{x_i, x_j \in V} f_{ij}(d_i, d_j) \quad , where\ x_i \leftarrow d_i,$$
$$x_j \leftarrow d_j\ in\ \mathcal{A}$$

For example, in Figure 2.2, $F(\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0)\}) = 4$ and $F(\{(x_1, 1),$ $(x_2, 1)$, $(x_3, 1)$, $(x_4, 1)\}) = 0$. In this example, $\mathcal{A}^* = \{(x_1, 1)$,$(x_2, 1)$, $(x_3, 1)$, $(x_4, 1)\}$.

Returning to the sensor network domain described in section 2.1, we may cast each sensor as a variable and the three radar heads as variable values. Constraints are derived from the geographic relationship between sensors. In particular, if a sensor activates one of its radar heads and detects a target, constraints require nearby agents to activate their corresponding radar head to help track the detected target. If a nearby agent violates the constraint, e.g. it does not activate the corresponding radar head because it is busy tracking another target, the agents pay a cost equal to the weight of the untracked target. We will examine this domain and its representation in the DCOP framework in more detail in Chapter 5.

We are interested primarily in managing interdependencies between different agents' choices. We will assume each agent is assigned a single variable and use the terms "agent" and "variable" interchangeably. Since agents sometimes have complex local problems, this is an assumption to be relaxed in future work. Yokoo et al. [52] describe some methods for dealing with multiple variables per agent in DisCSP and such methods may also apply to DCOP. We will also assume that constraints are binary, i.e., are defined over two variables. Note that generalization to n-ary constraints has been achieved in the DisCSP case without significant revisions to algorithms that were originally developed for binary constraints. Section 3.5 discusses ways in which the assumption of single variable per agent and limitation to binary constraints may be relaxed. We assume that neighbors are able to communicate. Messages, if received, are received in the order in which they are sent between any pair of agents. Messages sent from different agents to a single agent may be received in any order.

The computational complexity of DCOP as defined above is NP-hard. To see this, realize that 3-colorability of a graph, which is known to be NP-complete [31], can be modeled as an instance of DCOP: Assign all graphs nodes to a single agent, model graph edges as cost functions that return 0 if colors are different and 1 if colors are the same, and ask whether a solution of global cost 0 exists.

2.3.1 Scope of Model

The algorithmic techniques to be described later apply to a larger class of problems beyond summation over natural numbers. In fact, we can generalize to other "aggregation operators" such as minimization, which takes a set of natural numbers and returns the minimium in the set. In fact, our techniques can be applied to any associative, commutative, monotonic non-decreasing aggregation operator defined over a totally ordered set of valuations, with minimum and maximum element. This class of optimization functions is described formally by Schiex, Fargier and Verfaillie as Valued CSPs [37].

While the DCOP represention presented above is rather general, it is unable to model certain optimization functions. For example, we could not represent functions that are not monotonic, such as summation over integers where negative numbers may cause a decrease in the value of the optimization function. Also, the assumption that the global cost function can be decomposed into the aggregation of binary cost functions can be limiting in representing some optimization functions, although we will discuss ways to deal with this problem in later sections (such as dualization of representations and the generalization of our methods to n-ary constraints).

Chapter 3

Asynchronous Complete Method for DCOP

In this chapter we outline a novel lower-bound based distributed search method for DCOP. The first section outlines the three key features we desire in our solution, analyzes why existing methods fail to provide them and describes the basic ideas behind the new method. The following sections present algorithm details, theoretical proofs of completeness and experimental evaluations.

3.1 Basic Ideas

DCOP demands techniques that go beyond existing methods for finding distributed satisfactory solutions and their simple extensions for optimization. A DCOP method for the types of real-world applications previously mentioned must meet three key requirements. First, since the problem is inherently distributed, we require a method where agents can optimize a global function in a distributed fashion using only local communication (communication with neighboring agents). Methods were all agents must communicate with a single central agent are unacceptable. Second, we require a method that is able to find solutions quickly by allowing agents to operate asynchronously. A synchronous method where an agent sits idle while waiting for a particular message from a particular agent is unacceptable because it is wasting time when it could potentially be doing useful work. For example, Figure 3.1 shows groups of loosely connected agent subcommunities which could potentially execute search in parallel rather than sitting idle. Finally, provable quality guarantees on system performance are needed. For example, mission failure by a satellite constellation performing space exploration can result in extraordinary monetary and scientific losses. Thus, we require a method that efficiently finds provably optimal solutions whenever possible and also allows principled solution-quality/computation-time tradeoffs when time is limited.

A solution strategy that is able to provide quality guarantees, while at the same time meeting the requirements of distributedness and asynchrony, is currently missing from the research literature. A well-known method for solving DisCSP is the Asynchronous Backtracking (ABT) algorithm of Yokoo, Durfee, Isida, and Kuwabara [50]. Simple extensions of ABT for optimization have relied on converting an optimization problem into a sequence of satisfaction problems in order to allow the use of a DisCSP algorithm [17]. This approach has applied only to limited types of optimization problems (e.g.

Hierarchical DisCSPs, Maximal DisCSPs), but has failed to apply to general DCOP problems, even rather natural ones such as minimizing the total number of constraint violations (MaxCSP). Other existing algorithms that provide quality guarantees for optimization problems, such as the Synchronous Branch and Bound (SynchBB) algorithm [16] discussed later, are prohibitively slow since they require synchronous, sequential communication. Other fast, asynchronous solutions, such as variants of local search [16] [53], cannot provide guarantees on the quality of the solutions they find.

As we can see from the above, one of the main obstacles for solving DCOP is combining quality guarantees with asynchrony. Previous approaches have failed to provide quality guarantees in DCOP using a distributed, asynchronous model because it is difficult to ensure a systematic backtrack search when agents are asynchronously changing their variable values. We argue that the main reason behind these failures is that previous approaches insist on backtracking *only* when they conclude, with certainty, that the current solution will not lead to the optimal solution. For example, an agent executing the ABT algorithm concludes with certainty that the current partial solution being explored will not lead to a global satisfactory solution whenever it locally detects an unsatisfiable constraint. Thus, while agents are able to asynchronously change variable values in ABT, it is only because of the limited representation of DisCSP, where only one constraint needs to be broken for a candidate solution to be globally inconsistent. Extensions of ABT for optimization problems [17] have continued to rely on a satisfaction-based representation and have failed to apply to general DCOP also for this reason. Similarly, the SynchBB algorithm concludes with certainty that the current partial solution will not lead to a globally optimal solution whenever its cost exceeds a global upper bound. This approach to DCOP fails to be asynchronous and parallel because computing a global upper bound requires that all costs in the constraint network be accumulated within a single agent before decisions can be made.

To alleviate the above difficulties, we present *Adopt* (Asynchronous Distributed Optimization), the first algorithm for DCOP that can find optimal solutions using only localized asynchronous communication and polynomial space at each agent. Communication is local in the sense that an agent does not send messages to every other agent, but only to neighboring agents. An assumption, to be relaxed later, is that communication is reliable.

A key idea behind Adopt is to obtain asynchrony by allowing each agent to change variable value whenever it detects there is a *possibility* that some other solution may be better than the one currently under investigation. This search strategy allows asynchronous computation because an agent does not need global information to make its local decisions – it can go ahead and begin making decisions with only local information. The three main ideas in Adopt are described next.

Lower-bound Based Search. Adopt performs distributed backtrack search using an ”opportunistic” best-first search strategy, i.e., each agent keeps on choosing the best

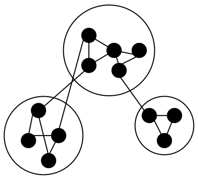


Figure 3.1: Loosely connected subcommunities of problem solvers

value based on the current available information. Stated differently, each agent always chooses the variable value with smallest lower bound. This search strategy is in contrast to previous distributed "branch and bound" type search algorithms for DCOP (e.g. SynchBB [16]) that require agents to change value only when cost exceeds a global upper bound (which proves that the current solution must be suboptimal). Adopt's new search strategy is significant because lower bounds are more suitable for asynchronous search – a lower bound can be computed without necessarily having accumulated global cost information. In Adopt, an initial lower bound is immediately computed based only on local cost. The lower bound is then iteratively refined as new cost information is asynchronously received from other agents. Note that because this search strategy allows agents to abandon partial solutions before they have proved the solution is definitely suboptimal, they may be forced to reexplore previously considered solutions. The next idea in Adopt addresses this issue.

Backtrack Thresholds. To allow agents to efficiently reconstruct a previously explored solution, which is a frequent action due to Adopt's search strategy, Adopt uses the second idea of using a stored lower bound as a *backtrack threshold*. This technique increases efficiency, but requires only polynomial space in the worst case, which is much better than the exponential space that would be required to simply memorize partial solutions in case they need to be revisited. The basic idea behind backtrack thresholds is that if a parent agent knows from previous search experience that lb is a lower bound on cost within one of its subtrees, it should inform the subtree not to bother searching for a solution whose cost is less than lb when a partial solution is revisited. In this way, a parent agent calculates backtrack threshold using a previously known lower bound and sends the threshold to its children. Then, the child uses the backtrack threshold as an *allowance* on solution cost – a child agent will not change its variable value so long as the cost of the current solution is less than the given backtrack threshold. Since the backtrack threshold is calculated using a previously known lower bound, it is ensured to be less than or equal to the cost of the optimal solution. Thus, we know the optimal solution will not be missed.

To make the backtrack threshold approach work when multiple subcommunities search in parallel, a parent agent x_p must distribute its cost allowance, denoted $threshold$, correctly to its multiple children. This is a challenging task because the parent does not remember how cost was accumulated from its children in the past (to do so would require exponential space in the worst case). We address this difficulty in the following way. If x_p chooses variable value d which has a local cost of $\delta(d)$, it subdivides the remaining allowance, $threshold - \delta(d)$, arbitrarily among its children. However, this subdivision may be incorrect, and so must be corrected over time. Let x_i be a child of

x_p. After some search, x_i may discover that its portion of $threshold$, denoted $t(d, x_i)$, is too low because the lower bound on the cost in its subcommunity, denoted $lb(d, x_i)$, exceeds $t(d, x_i)$. When this happens, x_i unilaterally raises its own allowance and reports $lb(d, x_i)$ to its parent x_p. The parent agent then redistributes $threshold$ among its children by increasing $t(d, x_i)$ and decreasing the portions given to the other children. Informally, the parent maintains an **AllocationInvariant** (described later) which states that its local cost plus the sum of $t(d, x_i)$ over all children x_i must equal its backtrack threshold $threshold$ and a **ChildThresholdInvariant**, which states that no child x_i should be given allowance $t(d, x_i)$ less than its lower bound $lb(d, x_i)$. Using these invariants (and cost feedback from its children), the parent continually re-balances the subdivision of backtrack threshold among its children until the correct threshold is given to each child.

Termination Detection. Finally, the third key idea is the use of bound intervals for tracking the progress towards the optimal solution, thereby providing a built-in termination detection mechanism. A bound interval consists of both a lower bound and an upper bound on the optimal solution cost. When the size of the bound interval shrinks to zero, i.e., the lower bound equals the upper bound, the cost of the optimal solution has been determined and agents can safely terminate when a solution of this cost is obtained. Most previous distributed search algorithms have required a separate termination detection algorithm. In contrast, the bound intervals in Adopt provide a natural

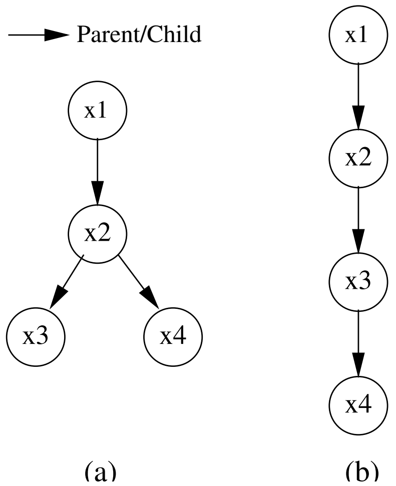


Figure 3.2: Two valid tree orderings for the constraint graph in figure 2.2

termination detection criterion integrated within the algorithm. This is a significant advance because bound intervals can be used to perform bounded-error approximation. As soon as the bound interval shrinks to a user-specified size, agents can terminate early while guaranteeing they have found a solution whose cost is within the given distance of the optimal solution. This means that agents can find an approximate solution faster than the optimal one but still provide a theoretical guarantee on global solution quality.

3.2 Adopt Algorithm

Tree Ordering. Before executing Adopt, agents are ordered into a Depth-First Search (DFS) tree. Thus, unlike previous algorithms such as SynchBB [16], Adopt does *not* require a linear ordering on all the agents. A DFS tree order is defined by directed *parent-child* edges. A valid DFS tree requires that the graph formed by parent-child

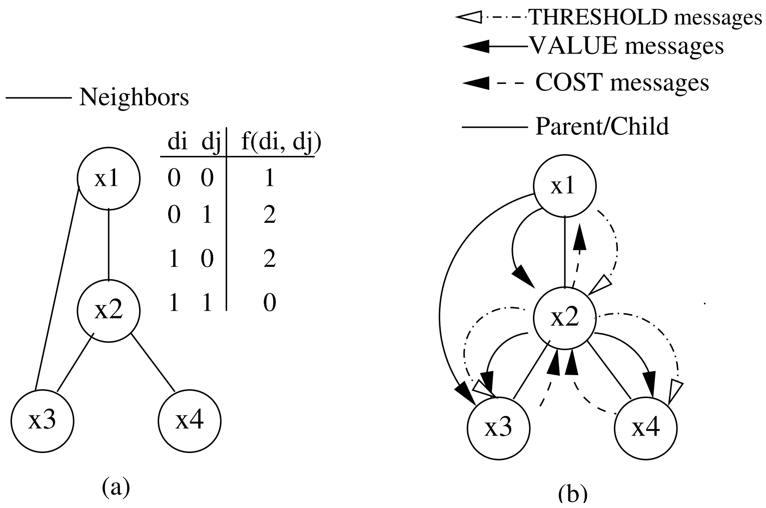


Figure 3.3: (a) Constraint graph. (b) Communication graph.

edges is acyclic. A single agent is designated as the root and all other agents have a single parent. Note that an agent may be the parent of multiple *children* but no agent may have multiple parents. We will not describe the details of the algorithm for constructing this tree order. It has been shown that a valid tree order can be constructed in polynomial-time in a distributed fashion[26]. Yokoo et al. [50] describe a method where a tree order can be constructed under the reasonable assumption that each agent has a unique identifier. For our purposes, we will assume the DFS ordering is done in a preprocessing step.

The given input DCOP constraint graph places one important restriction on the space of valid tree orderings. Specifically, we require that there exist no constraint between two agents in different subtrees of the DFS tree ordering. The advantage of this restriction is that agents in different subtrees are able to search for solutions independently. However, it is important to understand that this restriction does *not* limit the space of input constraint graphs – every constraint graph can be ordered into a valid DFS tree. As an example consider a linear (total) ordering where no agent has more than one single child. This is a valid tree ordering for any given constraint graph because there are no branches in the tree. Thus, there cannot exist a constraint between agents in different subtrees.

For a given input DCOP constraint graph, there may be many valid tree orderings. Figure 3.2 shows two possible DFS tree orderings for the constraint graph in Figure 2.2. In Figure 3.2.a, x_1 is the root, x_1 is the parent of x_2, and x_2 is the parent of both x_3 and x_4. Note that constraints are allowed between an agent and any of its ancestors or descendents (there is a constraint between x_1 and x_3). In Figure 3.2.b shows an alternative tree ordering where the agents are ordered in a linear order.

In this work, we do not address how distributed agents can choose the "best" tree ordering, although it is an important issue deserving further study. Researchers in centralized CSP have developed sophisticated heuristics for choosing good variable orders in backtrack style search[34]. A example includes the First Fail Principle which states that variables with smaller domain sizes should be instantiated first. The corresponding heuristic in our distributed situation would recommend that agents with smaller domain sizes to be higher in the tree. Another dimension to consider when comparing two tree orders is tree depth, i.e., the length of the longest path from root to leaf. Our intuition is that a tree with smaller depth is better because information is able to flow up the tree faster. For example, we should expect that Figure 3.2.a, where the tree depth is 3, is a better tree order than Figure 3.2.b, where the tree depth is 4. This intuition may become more obvious to the reader after the Adopt algorithm is explained.

Finally for simplicity, we will also assume that every parent and child are neighbors (a constraint exists between them). However, this is not a real restriction since we may always add a "dummy" zero-cost constraint between two agents who are chosen to be parent/child but are not neighbors in the input constraint graph.

Algorithm Overview. Once the DFS tree is constructed, each agent x_i concurrently executes the following algorithm.

- Initialize the lower bound for each value in D_i to zero. Assign a random value to your variable x_i.
- Send your current value of x_i to each neighbor lower in the DFS tree.
- When receive the value of a neighbor x_j, for each value D_i evaluate the constraint between x_i and x_j. Add the cost of the constraint to the lower bound for each of your values. If the lower bound for the current value is higher than the lower bound for some other value d, switch value to d.
- Send the lower bound for your value with least lower bound to your parent in the DFS tree. Attach the variable value of the parent under which this lower bound was computed as a "context".
- When receive a lower bound from your child attached with your value d, add the reported lower bound to the lower bound for d. If the lower bound for your current value is higher than the lower bound for some other value, switch to value with least lower bound.

- When a higher neighbor changes variable value, re-initialize the lower bound for each value in D_i to zero.
- Continue sending and receiving messages and changing values as dictated above until the following *termination condition* is true: The lower bound LB for one value d is also an upper bound, and the lower bound for all other values is higher than LB. Note that when this condition is true d is the globally optimal value for x_i until and unless a higher neighbor changes value.

Once the above termination condition is true at the root agent, the root sends a TERMINATE message to its children and terminates itself. After receiving a TERMINATE message, an agent knows that all of its higher neighbors have terminated. Once the termination condition is true at non-root agent x_i and it has received a TERMINATE message from its parent, x_i will send TERMINATE messages down to its children. In this way, TERMINATE messages are recursively sent down the tree until the termination condition is true at all agents and all agents have terminated.

3.2.1 Algorithm Details

The communication in Adopt is shown in Figure 3.3.b. The algorithm begins by all agents choosing their variable values concurrently. Variable values are sent down constraint edges via VALUE messages – an agent x_i sends VALUE messages only to neighbors lower in the DFS tree and receives VALUE messages only from neighbors higher in the DFS tree. A second type of message, a THRESHOLD message, is sent only from parent to child. A THRESHOLD message contains a single number representing a backtrack threshold, initially zero. Upon receipt of any type of message, an agent i) calculates cost and possibly changes variable value and/or modifies its backtrack threshold, ii) sends VALUE messages to its lower neighbors and THRESHOLD messages to its children and iii) sends a third type of message, a COST message, to its parent. A COST message is sent only from child to parent. A COST message sent from x_i to its parent contains the cost calculated at x_i plus any costs reported to x_i from its children. To summarize the communication, variable value assignments (VALUE messages) are sent down the DFS tree while cost feedback (COST messages) percolate back up the DFS tree. It may be useful to view COST messages as a generalization of NOGOOD message from DisCSP algorithms. THRESHOLD messages are sent down the tree to reduce redundant search.

Procedures from Adopt are shown in Figure 3.4 and 3.5. x_i represents the agent's local variable and d_i represents its current value.

- **Definition 1:** A *context* is a partial solution of the form $\{(x_j,d_j), (x_k,d_k)...\}$. A variable can appear in a context no more than once. Two contexts are *compatible* if they do not disagree on any variable assignment. $CurrentContext$ is a context which holds x_i's view of the assignments of higher neighbors.

A COST message contains three fields: $context$, lb and ub. The $context$ field of a COST message sent from x_l to its parent x_i contains x_l's $CurrentContext$. This field is necessary because calculated costs are dependent on the values of higher variables, so an agent must attach the context under which costs were calculated to every COST message. This is similar to the *context attachment* mechanism in ABT [50]. When x_i receives a COST message from child x_l, and d is the value of x_i in the $context$ field, then x_i stores lb indexed by d and x_l as $lb(d, x_l)$ (line 32). Similarly, the ub field is stored as $ub(d, x_l)$ and the $context$ field is stored as $context(d, x_l)$ (line 33-34). Before any COST messages are received or whenever contexts become incompatible, i.e., $CurrentContext$ becomes incompatible with $context(d, x_l)$, then $lb(d, x_l)$ is (re)initialized to zero and $ub(d, x_l)$ is (re)initialized to a maximum value Inf (line 3-4, 18-19, 29-30).

x_i calculates cost as local cost plus any cost feedback received from its children. Procedures for calculation of cost are not shown in Figure 3.4 but are implicitly given by procedure calls, such as **LB** and **UB**, defined next. The *local cost* at x_i, for a particular value choice $d_i \in D_i$, is the sum of costs from constraints between x_i and higher neighbors:

- **Definition 2:** $\delta(d_i) = \sum_{(x_j,d_j)\in CurrentContext} f_{ij}(d_i, d_j)$ is the *local cost* at x_i, when x_i chooses d_i.

For example, in Figure 3.3.a, suppose x_3 received messages that x_1 and x_2 currently have assigned the value 0. Then x_3's $CurrentContext$ would be $\{(x_1, 0), (x_2, 0)\}$. If x_3 chooses 0 for itself, it would incur a cost of 1 from $f_{1,3}(0, 0)$ (its constraint with

x_1) and a cost of 1 from $f_{2,3}(0, 0)$ (its constraint with x_2). So x_3's local cost, $\delta(0) = 1 + 1 = 2$.

When x_i receives a COST message, it adds $lb(d, x_l)$ to its local cost $\delta(d)$ to calculate a *lower bound for value* d, denoted $LB(d)$.

- **Definition 3:** $\forall d \in D_i, LB(d) = \delta(d) + \sum_{x_l \in Children} lb(d, x_l)$ is a *lower bound* for the subtree rooted at x_i, when x_i chooses d.

Similarly, x_i adds $ub(d, x_l)$ to its local cost $\delta(d)$ to calculate an *upper bound for value* d, denoted $UB(d)$.

- **Definition 4:** $\forall d \in D_i, UB(d) = \delta(d) + \sum_{x_l \in Children} ub(d, x_l)$ is a *upper bound* for the subtree rooted at x_i, when x_i chooses d.

The *lower bound for variable* x_i, denoted LB, is the minimum lower bound over all value choices for x_i.

- **Definition 5:** $LB = \min_{d \in D_i} LB(d)$ is a *lower bound* for the subtree rooted at x_i.

Similarly the *upper bound for variable* x_i, denoted UB, is the minimum upper bound over all value choices for x_i.

- **Definition 6:** $UB = \min_{d \in D_i} UB(d)$ is an *upper bound* for the subtree rooted at x_i.

x_i sends LB and UB to its parent as the lb and ub fields of a COST message (line 52). (Realize that LB need not correspond to x_i's current value, i.e., LB need not equal $LB(d_i)$). Intuitively, $LB = k$ indicates that it is not possible for the sum of the local costs at each agent in the subtree rooted at x_i to be less than k, given that all higher agents have chosen the values in $CurrentContext$. Similarly, $UB = k$ indicates that the optimal cost in the subtree rooted at x_i will be no greater than k, given that all higher agents have chosen the values in $CurrentContext$. Note that if x_i is a leaf agent, it does not receive COST messages, so $\delta(d) = LB(d) = UB(d)$ for all value choices $d \in D_i$ and thus, LB is always equal to UB in every COST message. If x_i is not a leaf but has not yet received any COST messages from its children, UB is equal to maximum value Inf and LB is the minimum local cost $\delta(d)$ over all value choices $d \in D_i$.

x_i's backtrack threshold is stored in the $threshold$ variable, initialized to zero (line 1). Its value is updated in three ways. First, its value can be increased whenever x_i determines that the cost of the optimal solution within its subtree must be greater than the current value of $threshold$. Second, if x_i determines that the cost of the optimal solution within its subtree must necessarily be less than the current value of $threshold$, it decreases $threshold$. These two updates are performed by comparing $threshold$ to LB and UB (lines 53-56, figure 3.5). The updating of $threshold$ is summarized by the following invariant.

- **ThresholdInvariant:** $LB \leq threshold \leq UB$. The threshold on cost for the subtree rooted at x_i cannot be less than its lower bound or greater than its upper bound.

A parent is also able to set a child's $threshold$ value by sending it a THRESHOLD message. This is the third way in which an agent's $threshold$ value is updated. The reason for this is that in some cases, the parent is able to determine a bound on the optimal cost of a solution within an agent's subtree, but the agent itself may not know this bound. The THRESHOLD message is a way for the parent to inform the agent about this bound.

A parent agent is able to correctly set the $threshold$ value of its children by allocating its own $threshold$ value to its children according to the following two equations. Let $t(d, x_l)$ denote the threshold on cost allocated by parent x_i to child x_l, given x_i chooses value d. Then, the values of $t(d, x_l)$ are subject to the following two invariants.

- **AllocationInvariant:** For current value $d_i \in D_i$, $threshold = \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$. The threshold on cost for x_i must equal the local cost of choosing d plus the sum of the thresholds allocated to x_i's children.

- **ChildThresholdInvariant:** $\forall d \in D_i, \forall x_l \in Children$, $lb(d, x_l) \leq t(d, x_l) \leq ub(d, x_l)$. The threshold allocated to child x_l by parent x_i cannot be less than the lower bound or greater than the upper bound reported by x_l to x_i.

By adhering to these invariants, an agent is able to use its own $threshold$ to determine bounds on the cost of the optimal solution within its childrens' subtrees.

The $threshold$ value is used to determine when to change variable value. Whenever $LB(d_i)$ exceeds $threshold$, x_i changes its variable value to one with smaller lower bound (line 40-41). (Such a value necessarily exists since otherwise ThresholdInvariant would be violated.) Note that x_i cannot prove that its current value is definitely suboptimal because it is possible that $threshold$ is less than the cost of the optimal solution. However, it changes value to one with smaller cost anyway – thereby realizing the best-first search strategy previously described.

3.2.2 Example of Algorithm Execution

Figure 3.6 shows an example of algorithm execution for the DCOP shown in figure 3.3. Line numbers mentioned in the description refer to figures 3.4 and 3.5. This example is meant to illustrate the search process and the exchange of VALUE and COST messages. COST messages are labelled in the figures as [LB,UB,CurrentContext]. For simplicity, not every message sent by every agent is shown. In particular, THRESHOLD messages are omitted from the description. (A later example will illustrate how backtrack thresholds are handled.)

All agents begin by concurrently choosing a value for their variable (line 5). For this example, let us assume they all choose value 0. Each agent sends its value to all

initialize
    $threshold \leftarrow 0$; $CurrentContext \leftarrow \{\}$
    forall $d \in D_i, x_l \in Children$ **do**
        $lb(d, x_l) \leftarrow 0$; $t(d, x_l) \leftarrow 0$
        $ub(d, x_l) \leftarrow Inf$; $context(d, x_l) \leftarrow \{\}$;
    enddo
    $d_i \leftarrow d$ that minimizes **LB**(d)
    backTrack

when received (**THRESHOLD**, t, $context$)
    if $context$ compatible with $CurrentContext$:
        $threshold \leftarrow t$
        maintainThresholdInvariant
        backTrack; **endif**

when received (**TERMINATE**, $context$)
    record TERMINATE received from parent
    $CurrentContext \leftarrow context$
    backTrack

when received (**VALUE**, (x_j,d_j))
    if TERMINATE not received from parent:
        add (x_j,d_j) to $CurrentContext$
        forall $d \in D_i, x_l \in Children$ **do**
            if $context(d, x_l)$ incompatible with $CurrentContext$:
                $lb(d, x_l) \leftarrow 0$; $t(d, x_l) \leftarrow 0$
                $ub(d, x_l) \leftarrow Inf$; $context(d, x_l) \leftarrow \{\}$; **endif; enddo**
        maintainThresholdInvariant
        backTrack; endif

when received (**COST**, x_k, $context$, lb, ub)
    $d \leftarrow$ value of x_i in $context$
    remove (x_i,d) from $context$
    if TERMINATE not received from parent:
        forall (x_j,d_j) $\in context$ and x_j is not my neighbor **do**
            add (x_j,d_j) to $CurrentContext$;**enddo**
        forall $d' \in D_i, x_l \in Children$ **do**
            if $context(d', x_l)$ incompatible with $CurrentContext$:
                $lb(d', x_l) \leftarrow 0$; $t(d', x_l) \leftarrow 0$
                $ub(d', x_l) \leftarrow Inf$; $context(d', x_l) \leftarrow \{\}$;**endif**;**enddo**;**endif**
    if $context$ compatible
    with $CurrentContext$:
        $lb(d, x_k) \leftarrow lb$
        $ub(d, x_k) \leftarrow ub$
        $context(d, x_k) \leftarrow context$
        maintainChildThresholdInvariant
        maintainThresholdInvariant;
    endif
    backTrack

Figure 3.4: Procedures for receiving messages in the Adopt algorithm. Definitions of terms **LB(d)**, **UB(d)**, **LB**, and **UB** are given in the text.

procedure backTrack

if $threshold ==$ **UB**:
  $d_i \leftarrow d$ that minimizes **UB(**d**)**
else if LB(d_i**)** $> threshold$:
  $d_i \leftarrow d$ that minimizes **LB(**d**)**;**endif**
SEND (**VALUE**, (x_i, d_i))
    to each lower priority neighbor
maintainAllocationInvariant
if $threshold ==$ **UB**:
  if TERMINATE received from parent
  or x_i is root:
    SEND (**TERMINATE**,
    $CurrentContext \cup \{(x_i, d_i)\}$)
      to each child
    Terminate execution; **endif;endif**
SEND (**COST**, x_i, $CurrentContext$, **LB**,**UB**)
  to parent

procedure maintainThresholdInvariant

if $threshold <$ **LB**
  $threshold \leftarrow$ **LB**; **endif**
if $threshold >$ **UB**
  $threshold \leftarrow$ **UB**; **endif**

%note: procedure assumes ThresholdInvariant is satisfied
procedure maintainAllocationInvariant

while $threshold > \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$ **do**
  choose $x_l \in Children$ where $ub(d_i, x_l) > t(d_i, x_l)$
  increment $t(d_i, x_l)$; **enddo**
while $threshold < \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$ **do**
  choose $x_l \in Children$ where $t(d_i, x_l) > lb(d_i, x_l)$
  decrement $t(d_i, x_l)$; **enddo**
SEND (**THRESHOLD**, $t(d_i, x_l)$, $CurrentContext$)
    to each child x_l

procedure maintainChildThresholdInvariant

forall $d \in D_i, x_l \in Children$ **do**
  while $lb(d, x_l) > t(d, x_l)$ **do**
    increment $t(d, x_l)$; **enddo;endo**
forall $d \in D_i, x_l \in Children$ **do**
  while $t(d, x_l) > ub(d, x_l)$ **do**
    decrement $t(d, x_l)$; **enddo;enddo**

Figure 3.5: Procedures in the Adopt algorithm (cont)

lower priority neighbors (figure 3.6.a). Since the algorithm is asynchronous, there are many possible execution paths from here – we describe one possible execution path.

x_2 will receive x_1's VALUE message. In line 15, it will record this value into its $CurrentContext$. In line 21, it will enter the **backTrack** procedure. x_2 computes $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 1 + 0 + 0 = 1$ and $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 2 + 0 + 0 = 2$. Since $LB(0) < LB(1)$, we have $LB = LB(0) = 1$. x_2 will also compute $UB(0) = \delta(0) + ub(0, x_3) + ub(0, x_4) = 1 + Inf + Inf = Inf$ and $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 2 + Inf + Inf = Inf$. Thus, $UB = Inf$. In line 38, $threshold$ is compared to UB. $threshold$ was set to 1 (in order to be equal to LB) in the maintainAllocationInvariant procedure call from line 20. Since $threshold = 1$ is not equal $UB = Inf$, the test fails. The test in line 40 also fails since $LB(0) = 1$ is not greater that $threshold = 1$. Thus, x_2 will stick with its current value $x_2 = 0$. In line 52, x_2 sends the corresponding COST message to x_1 (figure 3.6.b).

Concurrently with x_2's execution, x_3 will go through a similar execution. x_3 will evaluate its constraints with higher agents and compute $LB(0) = \delta(0) = f_{1,3}(0, 0) + f_{2,3}(0, 0) = 1 + 1 = 2$. A change of value to $x_3 = 1$ would incur a cost of $LB(1) = \delta(1) = f_{1,3}(0, 1) + f_{2,3}(0, 1) = 2 + 2 = 4$, so instead x_3 will stick with $x_3 = 0$. x_3 will send a COST message with $LB = UB = 2$, with associated context $\{(x_1, 0), (x_2, 0)\}$, to its parent x_2. x_4 executes similarly (figure 3.6.b).

Next, x_1 receives x_2's COST message. In line 31, x_1 will test the received context $\{(x_1, 0)\}$ against $CurrentContext$ for compatibility. Since x_1's $CurrentContext$ is empty, the test will pass. (Note that the root never receives VALUE messages, so its $CurrentContext$ is always empty.) The received costs will be stored in lines 32-33 as $lb(0, x_2) = 1$ and $ub(0, x_2) = Inf$. In line 37, execution enters the **backTrack** procedure. x_1 computes $LB(1) = \delta(1) + lb(1, x_2) = 0 + 0 = 0$ and $LB(0) = \delta(0) + lb(0, x_2) = 0 + 1 = 1$. Since $LB(1) < LB(0)$, we have $LB = LB(1) = 0$. Similarly, $UB = Inf$. Since $threshold = 0$ is not equal $UB = Inf$, the test in line 38 fails. The test in line 40 succeeds and x_1 will choose its value d that minimizes $LB(d)$. Thus, x_1 switches value to $x_1 = 1$. It will again send VALUE messages to its linked descendents (figure 3.6.c).

Next, let us assume that the COST messages sent to x_2 in figure 3.6.b are delayed. Instead, x_2 receives x_1's VALUE message from figure 3.6.c. In line 15, x_2 will update its $CurrentContext$ to $\{(x_1, 1)\}$. For brevity, the remaining portion of this procedure is not described.

Next, x_2 finally receives the COST message sent to it from x_3 in figure 3.6.b. x_2 will test the received context against $CurrentContext$ and find that they are incompatible because one contains $(x_1, 0)$ while the other contains $(x_1, 1)$ (line 31). Thus, the costs in that COST message will not be stored due to the context change. However, the COST message from x_4 will be stored in lines 32-33 as $lb(0, x_3) = 1$ and $ub(0, x_3) = 1$. In line

37, x_2 then proceeds to the **backTrack** procedure where it will choose its best value. The best value is now $x_2 = 1$ since $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 0 + 0 + 0$ and $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 2 + 0 + 1 = 3$. Figure 3.6.d shows the change in both x_2 and x_3 values after receiving x_1's VALUE message from figure 3.6.c. x_2 and x_3 send the new COST messages with the new context where $x_1 = 1$. x_2 also sends VALUE messages to x_3 and x_4 informing them of its new value.

Next, figure 3.6.e shows the new COST message that is sent by x_2 to x_1 after receiving the COST messages sent from x_3 and x_4 in figure 3.6.d. Notice that x_2 computes LB as $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 0 + 0 + 0$ and UB as $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 0 + 2 + 1 = 3$. Figure 3.6.e also shows the new COST message sent by x_3 after receiving x_2's new value of $x_2 = 1$. Similarly, x_4 will change variable value and send a COST message with $LB = 0$ and $UB = 0$. In this way, we see the agents have ultimately settled on the optimal configuration with all values equal to 1 (total cost = 0).

Finally in figure 3.6.f, x_2 receives the COST messages from figure 3.6.e, computes a new bound interval $LB = 0, UB = 0$ and sends this information to x_1. Upon receipt of this message, x_1 will compute $UB = UB(0) = \delta(0) + ub(0, x_2) = 0 + 0 = 0$. Note that x_1's $threshold$ value is also equal to zero. $threshold$ was initialized to zero in line 1 and can only be increased if i) a THRESHOLD message is received (line 8), or b) the ThresholdInvariant is violated (line 54, figure 3.5). The root never receives

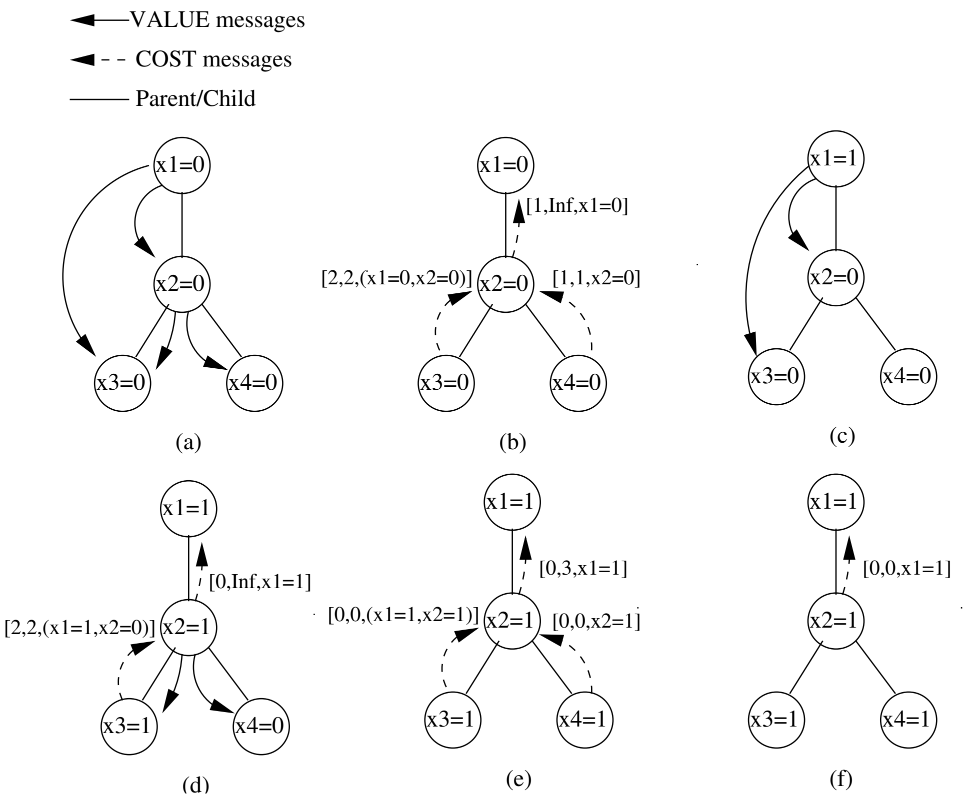


Figure 3.6: Example Adopt execution for the DCOP shown in figure 3.3

THRESHOLD messages, so case (i) never occurred. Since x_1's LB was never greater than zero in this example, $threshold$ could never have been less than LB, so case (ii) never occurred. Thus, $threshold$ was never increased and remains equal to zero. So, we have the test $threshold == UB$ in line 45 evaluate to true. In line 48, it will send a TERMINATE message to x_2, and then x_1 will terminate in line 51. x_2 will receive the TERMINATE message in line 11, evaluate $threshold == UB (= 0)$ to be true in line 45 and then terminate in line 51. The other agents will terminate in a similar manner.

3.2.3 Example of Backtrack Thresholds

We illustrate how backtrack thresholds are computed, updated and balanced between children. The key difficulty is due to context changes. An agent only stores cost information for the current context. When the context changes, the stored cost information must be deleted (in order to maintain polynomial space). If a previous context is later returned to, the agent no longer has the previous context's detailed cost information available. However, the agent had reported the total sum of costs to its parent, who has that information stored. So, although the precise information about how the costs were accumulated from the children is lost, the total sum is available from the parent. It is precisly this sum that the parent sends to the agent via the THRESHOLD message. The child then heuristically re-subdivides, or allocates, the threshold among its own children. Since this allocation may be incorrect, it then corrects for over-estimates over time as cost feedback is (re)received from the children.

Figure 3.7 shows a portion of a DFS tree. The constraints are not shown. Line numbers mentioned in the description refer to figure 3.4 and figure 3.5. x_p has parent x_q, which is the root, and two children x_i and x_j. For simplicity, assume $D_p = \{d_p\}$ and $\delta(d_p) = 1$, i.e, x_p has only one value in its domain and this value has a local cost of 1.

Suppose x_p receives COST messages containing lower bounds of 4 and 6 from its two children (figure 3.7.a). The costs reported to x_p are stored as $lb(d_p, x_i) = 4$ and

$lb(d_p, x_j) = 6$ (line 32) and associated context as $context(d_p, x_i) = context(d_p, x_j) = \{(x_q, d_q)\}$. LB is computed as $LB = LB(d_p) = \delta(d_p) + lb(d_p, x_i) + lb(d_p, x_j) = 1 + 4 + 6 = 11$. In figure 3.7.b, the corresponding COST message is sent to parent x_q. After the COST message is sent, suppose a context change occurs at x_p through the receipt of a VALUE message $x_q = d'_q$. In line 18-19, x_p will reset $lb(d_p, x_i)$, $lb(d_p, x_j)$, $t(d_p, x_i)$ and $t(d_p, x_j)$ to zero.

Next, x_q receives the information sent by x_p. x_q will set $lb(d_q, x_p) = 11$ (line 32), and enter the **maintainChildThresholdInvariant** procedure (line 35). Let us assume that $t(d_q, x_p)$ is still zero from initialization. Then, the test in line 65 succeeds since $lb(d_q, x_p) = 11 > t(d_q, x_p) = 0$ and x_q detects that the ChildThresholdInvariant is being violated. In order to correct this, x_q increases $t(d_q, x_p)$ to 11 in line 66.

Next, in figure 3.7.c, x_q revisits the value d_q and sends the corresponding VALUE message $x_q = d_q$. Note that this solution context has already been explored in the past, but x_p has retained no information about it. However, the parent x_q has retained the sum of the costs, so x_q sends the THRESHOLD message with $t(d_q, x_p) = 11$.

Next, x_p receives the THRESHOLD message. In line 8, the value is stored in the $threshold$ variable. Execution proceeds to the **backTrack** procedure where **maintainAllocationInvariant** is invoked in line 44. Notice that the test in line 57 of **maintainAllocationInvariant** evaluates to true since $threshold = 11 > \delta(d_p) + t(d_p, x_i) + t(d_p, x_j) = 1 + 0 + 0$. Thus, in lines 57-59, x_p increases the thresholds for its children until the invariant is satisfied. Suppose that the split is $t(d_p, x_i) = 10$ and $t(d_p, x_j) = 0$. This is an arbitrary subdivision that satisfies the AllocationInvariant – there are many other values of $t(d_p, x_i)$ and $t(d_p, x_j)$ that could be used. In line 63, these values are sent via a THRESHOLD message (figure 3.7.d).

By giving x_i a threshold of 10, x_p risks sub-optimality by overestimating the threshold in that subtree. This is because the best known lower bound in x_i's subtree was only 4. We now show how this arbitrary allocation of threshold can be corrected over time. Agents continue execution until, in figure 3.7.e, x_p receives a COST message from its right child x_j indicating that the lower bound in that subtree is 6. x_j is guaranteed to send such a message because there can be no solution in that subtree of cost less than 6, as evidenced by the COST message previously sent by x_j in figure 3.7.a. x_p will set $lb(d_p, x_j) = 6$ (line 32) and enter the **MaintainChildThresholdInvariant** procedure in line 35. Note that the test in line 65 will succeed since $lb(d_p, x_j) = 6 > t(d_p, x_j) = 5$ and the ChildThresholdInvariant is being violated. In order to correct this, x_p increases $t(d_p, x_j)$ to 6 in line 66. Execution returns to line 35 and continues to line 44, where the **maintainAllocationInvariant** is invoked. The test in line 60 of this procedure will succeed since $threshold = 11 < \delta(d_p) + t(d_p, x_i) + t(d_p, x_j) = 1 + 10 + 6 = 17$ and so the AllocationInvariant is being violated. In lines 60-62, x_p lowers $t(d_p, x_i)$ to 4 to satisfy the invariant. In line 63, x_p sends the new (correct) threshold values to its children (figure 3.7.f).

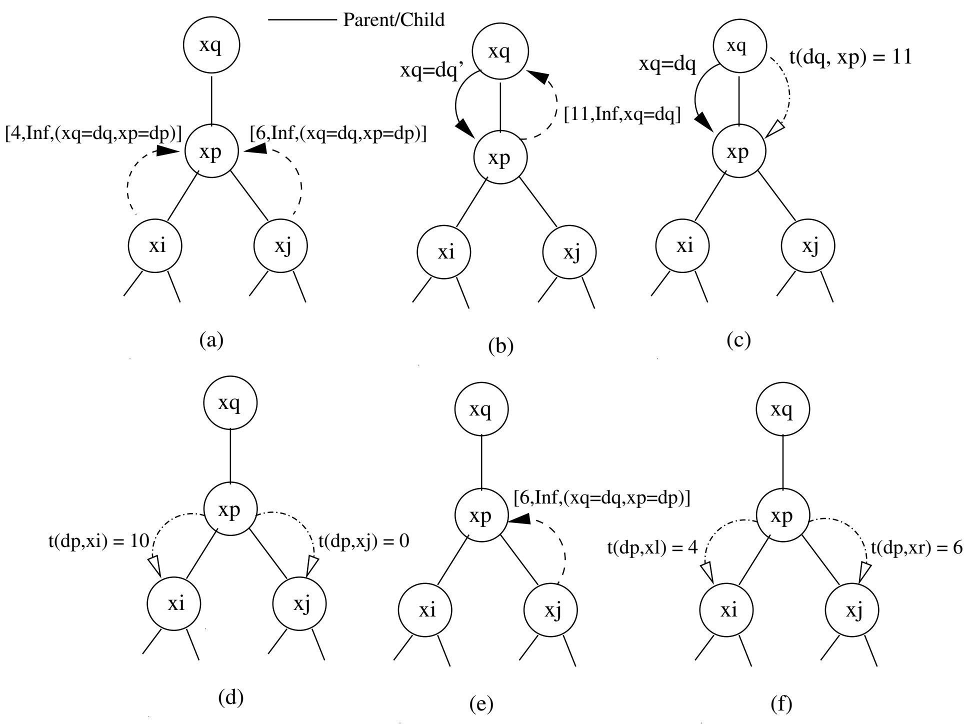


Figure 3.7: Example of backtrack thresholds in Adopt

In this way, a parent agent continually rebalances the threshold given to its independent subtrees in order to avoid overestimating the cost in each subtree while allowing more efficient search.

3.3 Correctness and Complexity

In order to show that Adopt is correct and complete, we must first show that the lower bounds and upper bounds computed at each agent are always correct. Thus, Theorem 1 shows that the lower bound LB computed by an agent is *never greater* than the cost of the optimal solution within its subtree, and the upper bound UB is *never less* than the cost of the optimal solution within its subtree. The proof of Theorem 1 relies on the following observation: if x_i chooses some value d, then the cost of the best solution possible in the subtree rooted at x_i is equal to (a) the local cost at x_i for value d *plus* (b) the cost of the optimal solution in the subtrees rooted at x_i's children given that x_i has chosen value d. Therefore, the optimal solution in the subtree rooted at x_i is obtained if and only if x_i chooses a value that minimizes this total cost. To state this observation formally, we need to define the following term: let $OPT(x_i, context)$ denote the cost of the optimal solution in the subtree rooted at x_i, given that higher priority variables have values in $context$. For example, if x_i is a leaf, then $OPT(x_i, context) = \min_{d \in D_i} \delta(d)$, i.e., the cost of the optimal solution in the subtree rooted at a leaf (which is a single-node tree consisting of only the leaf) is the value that minimizes the local cost at the leaf. We now state Property 1.

Property 1: $\forall x_i \in V$,

$$OPT \quad (x_i, CurrentContext) \stackrel{def}{=} \min_{d \in D_i} \delta(d) +$$

$$\sum_{x_l \in Children} OPT(x_l, CurrentContext \cup (x_i, d))$$

The proof of Theorem 1 proceeds by induction. The base case follows from the fact $LB = OPT(x_i\ , CurrentContext) = UB$ is always true at a leaf agent. The inductive hypothesis assumes that LB (UB) sent by x_i to its parent is never greater (less) than the cost of the optimal solution in the subtree rooted at x_i. The proof also relies on the fact that costs are reported to only one parent so there is no double counting of costs.

Theorem 1 $\forall x_i \in V$,

$$LB \leq OPT(x_i, CurrentContext) \leq UB$$

Proof: By induction on agent ordering, leaf to root.

Base Case I: x_i is a leaf. Since x_i has no children, the equations for LB and UB (see Definition 4 and 5 in section 3.2.1) simplify to $LB = \min_{d \in D_i} \delta(d) = UB$. Property 1 simplifies to $OPT(x_i, CurrentContext) = \min_{d \in D_i} \delta(d)$ for the same reason. So we conclude $LB = \min_{d \in D_i} \delta(d) = OPT(x_i, CurrentContext) = UB$. Done.

Base Case II: Every child of x_i is a leaf. We will show $LB \leq OPT(x_i, CurrentContext)$. The proof for $OPT(x_i, CurrentContext) \leq UB$ is analogous.

Since all children x_l are leaves, we know from Base Case I that $lb(d, x_l) \leq OPT(x_l, CurrentContext \cup (x_i, d))$. Furthermore, each child x_l sends COST messages only to x_i, so costs are not double-counted. We substitute $OPT(x_l, CurrentContext \cup (x_i, d))$ for $lb(d, x_l)$ into the definition of LB to get the following:

$$LB = \min_{d \in D_i} \delta(d) + \sum_{x_l \in Children} lb(d, x_l) \leq$$
$$\min_{d \in D_i} \delta(d) + \sum_{x_l \in Children} OPT(x_l, CurrentContext \cup (x_i, d))$$

Now we can simply substitute Property 1 into the above to get

$$LB \leq OPT(x_i, CurrentContext)$$

and we are done.

Inductive Hypothesis: $\forall d \in D_i, \forall x_l \in Children$,

$$lb(d, x_l) \leq OPT(x_l, CurrentContext \cup (x_i, d)) \leq ub(d, x_l)$$

The proof of the general case is identical to that of Base Case II, except we assume $lb(d, x_l) \leq OPT(x_l, CurrentContext \cup (x_i, d))$ from the Inductive Hypothesis, rather than from the assumption that x_l is a leaf. $\Box$

Next, we must show that Adopt will eventually terminate. Adopt's termination condition is shown in line 45 of Figure 3.4, namely the condition $threshold = UB$ must hold, and if x_i is not the root a TERMINATE message must also be received from the parent. In Theorem 2, we show that if the $CurrentContext$ is fixed, then $threshold = UB$ will eventually occur. The proof follows from the fact that agents continually receive cost reports LB and UB from their children and pass costs up to their parent. Theorem 1 showed that LB has an upper bound and UB has a lower bound, so LB must eventually stop increasing and UB must eventually stop decreasing.

The ThresholdInvariant forces $threshold$ to stay between LB and UB until ultimately $threshold = UB$ occurs.

Theorem 2 *$\forall x_i \in V$, if $CurrentContext$ is fixed, then $threshold = UB$ will eventually occur.*

Proof: By induction on agent priorities, leaf to root.

Base Case: x_i is a leaf. $LB = UB$ is always true at x_i because it is a leaf. Every agent maintains the ThresholdInvariant $LB \leq threshold \leq UB$. So $threshold = UB$ must always be true at a leaf.

Inductive Hypothesis: If $CurrentContext$ is fixed and x_i fixes its variable value to d_i, then $\forall x_l \in Children$, $threshold = UB$ will eventually occur at x_l and it will report an upper bound ub via a COST message, where $ub = t(d_i, x_l)$.

Assume $CurrentContext$ is fixed. To apply the Inductive Hypothesis, we must show that x_i will eventually fix its variable value. To see this, note that x_i changes its variable value only when $LB(d_i)$ increases. By Theorem 1, LB is always less than the cost of the optimal solution. LB cannot increase forever and so x_i must eventually stop changing its variable value. We can now apply the Inductive Hypothesis which says that when x_i fixes its value, each child will eventually report an upper bound $ub = t(d_i, x_l)$. This means $t(d_i, x_l) = ub(d_i, x_l)$ will eventually be true at x_i. We can substitute $t(d_i, x_l)$ for $ub(d_i, x_l)$ into the definition of UB to get the following:

$$UB \overset{def}{\leq} UB(d_i) \overset{def}{=} \delta(d_i) + \sum_{x_l \in Children} ub(d_i, x_l)$$
$$= \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$$

Using the AllocationInvariant $threshold = \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$, we substitute $threshold$ into the above to get $UB \leq threshold$. The right-hand side of the ThresholdInvariant states $threshold \leq UB$. So we have both $UB \leq threshold$ and $threshold \leq UB$. So $threshold = UB$ must be true and the Theorem is proven. $\square$

Note that the algorithm behaves differently depending on whether x_i's $threshold$ is set below or above the cost of the optimal solution. If $threshold$ is less than the cost of the optimal solution, then when LB increases above $threshold$, x_i will raise $threshold$ until ultimately, $LB = threshold = UB$ occurs. On the other hand, if $threshold$ is greater than the cost of the optimal solution, then when UB decreases below $threshold$, x_i will lower $threshold$ so $threshold = UB$ occurs. In the second case, LB may remain less than UB at termination since some variable values may not be re-explored.

Theorem 2 is sufficent to show algorithm termination because the root has an fixed (empty) $CurrentContext$ and will therefore terminate when $threshold = UB$ occurs. Before it terminates, it sends a TERMINATE message to its children informing them of its final value (line 48). It is clear to see that when a TERMINATE message is received from the parent, an agent knows that its current context will no longer change since all higher agents have already terminated.

From Theorem 1, if the condition $threshold = UB$ occurs at x_i, then there exists at least one solution within the subtree rooted at x_i whose cost is less than or equal $threshold$. From Theorem 2, the condition $threshold = UB$ necessarily occurs. Next, Theorem 3 shows that the final value of $threshold$ is equal to the cost of the optimal solution.

Theorem 3 *$\forall x_i \in V$, x_i's final $threshold$ value is equal to $OPT(x_i, CurrentContext)$.*

Proof: By induction on agent priorities, root to leaf.

Base Case: x_i is the root. The root terminates when its (final) $threshold$ value is equal UB. $LB = threshold$ is always true at the root because $threshold$ is initialized to zero and is increased as LB increases. The root does not receive THRESHOLD messages so this is the only way $threshold$ changes. We conclude $LB = threshold = UB$ is true when the root terminates. This means the root's final $threshold$ value is the cost of a global optimal solution.

Inductive Hypothesis: Let x_p denote x_i's parent. x_p's final $threshold$ value is equal to $OPT(x_p, CurrentContext)$.

We proceed by contradiction. Suppose x_i's final threshold is an overestimate. By the inductive hypothesis, x_p's final threshold is not an overestimate. It follows from the AllocationInvariant that if the final threshold given to x_i (by x_p) is too high, x_p must have given some other child (a sibling of x_i), say x_j, a final threshold that is too low (See Figure 3.7). Let d denote x_p's current value. Since x_j's threshold is too low, it will be unable to find a solution under the given threshold and will thus increase its own threshold. It will report lb to x_p, where $lb > t(d, x_j)$. Using Adopt's invariants, we can conclude that $threshold = UB$ cannot be true at x_p, so x_p cannot have already terminated. By the ChildThresholdInvariant, x_p will increase x_j's threshold so that $lb(d, x_j) \leq t(d, x_j)$. Eventually, $lb(d, x_j)$ will reach an upper bound and $lb(d, x_j) = t(d, x_j) = ub(d, x_j)$ will hold. This contradicts the statement that x_j's final threshold is too low. By contradiction, x_j's final threshold value cannot be too low and x_i's final threshold cannot be too high. □

The worst-case time complexity of Adopt is exponential in the number of variables n, since constraint optimization is known to be NP-hard. To determine the worst-case space complexity at each agent, note that an agent x_i needs to maintain a $CurrentContext$ which is at most size n, and an $lb(d, x_l)$ and $ub(d, x_l)$ for each domain value and child, which is at most $\mid D_i \mid \times n$. The $context(d, x_l)$ field can require n^2 space in the worst case. Thus, we can say the worst-case space complexity of Adopt is polynomial in the number of variables n. However, it can be reduced to linear at the potential cost of efficiency. Since $context(d, x_l)$ is always compatible with $CurrentContext$, $CurrentContext$ can be used in the place of each $context(d, x_l)$, thereby giving a space complexity of $\mid D_i \mid \times n$. This can be inefficient since an agent must reset all $lb(d, x_l)$ and $ub(d, x_l)$ whenever $CurrentContext$ changes, instead of only when $context(d, x_l)$ changes.

3.4 Evaluation

As in previous experimental set-ups[17], we experiment on distributed graph coloring with 3 colors. One node is assigned to one agent who is responsible for choosing its color. Cost of solution is measured by the total number of violated constraints. We will experiment with graphs of varying *link density* – a graph with link density d has dn links, where n is the number of nodes in the graph. For statistical signficance, each datapoint representing number of cycles is the average over 25 random problem instances. The randomly generated instances were not explicitly made to be overconstrained, but note that link density 3 is beyond phase transition, so randomly generated graphs with this link density are almost always overconstrained. The tree-structured DFS priority ordering for Adopt was formed in a preprocessing step. To compare Adopt's performance with algorithms that require a chain (linear) priority ordering, a depth-first traversal of Adopt's DFS tree was used.

As in [17], we measure "time to solution" in terms of synchronous cycles. One *cycle* is defined as all agents receiving all incoming messages and sending all outgoing messages simultaneously. Other evaluation metrics such as "wall clock" time are

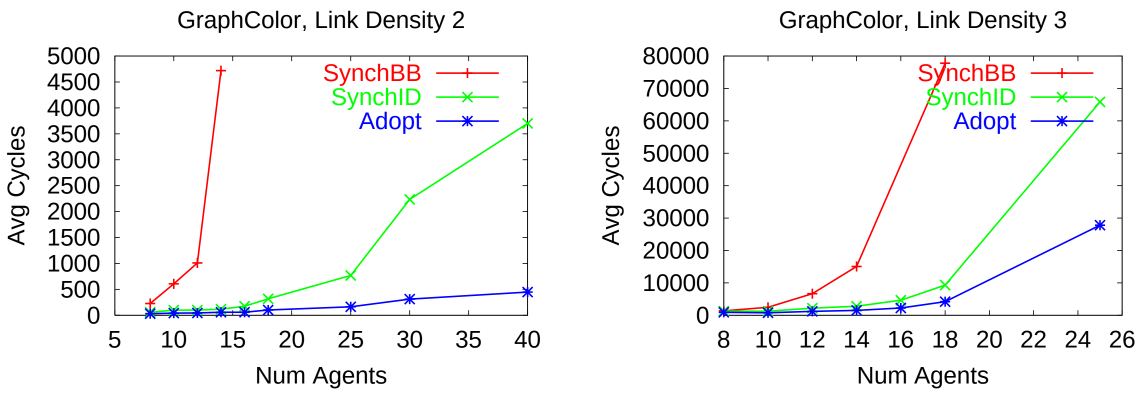


Figure 3.8: Average number of cycles required to find the optimal solution (MaxCSP)

very sensitive to variations in computation speeds at different agents or communication delays between agents. These factors are often unpredictable and we would like to control for them when performing systematic experiments. The synchronous cycle metric allows repeatable experiments and controlled comparisons between different asynchronous algorithms because it is not sensitive to differing computation speeds at different agents or fluctuations in message delivery time.

We present the empirical results from experiments using three different algorithms for DCOP – Synchronous Branch and Bound (SynchBB), Synchronous Iterative Deepening (SynchID) and Adopt. We illustrate that Adopt outperforms SynchBB[16], a distributed version of branch and bound search and the only known algorithm for DCOP that provides optimality guarantees. In addition, by comparing with SynchID we show that the speed-up comes from two sources: a) Adopt's novel search strategy, which uses

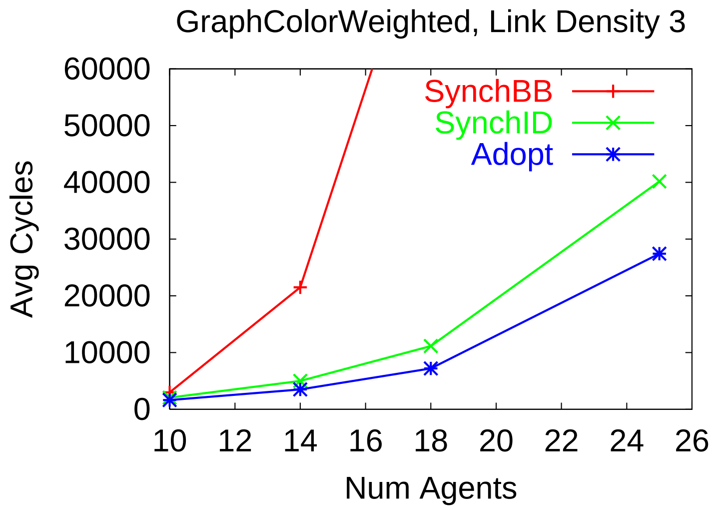


Figure 3.9: Average number of cycles required to find the optimal solution (Weighted CSP)

lower bounds instead of upper bounds to do backtracking, and b) the asynchrony of the algorithm, which enables concurrency.

SynchID is an algorithm we have constructed in order to isolate the causes of speed-ups obtained by Adopt. SynchID simulates iterative deepening search[21] in a distributed environment. SynchID's search strategy is similar to Adopt since both algorithms iteratively increase lower bounds and use the lower bounds to do backtracking. However, the difference is that SynchID maintains a single global lower bound and agents are required to execute sequentially and synchronously while in Adopt, each agent maintains its own lower bound and agents are able to execute concurrently and asynchronously. In SynchID, the agents are ordered into a linear chain. (A depth-first traversal of Adopt's DFS tree was used in our experiments.) The highest priority agent chooses a value for its variable first and initializes a global lower bound to zero. The next agent in the chain attempts to extend this solution so that the cost remains under the lower bound. If an agent finds that it cannot extend the solution so that the cost is less than the lower bound, a backtrack message is sent back up the chain. Once the highest priority agent receives a backtrack message, it increases the global lower bound and the process repeats. In this way, agents synchronously search for the optimal solution by backtracking whenever the cost exceeds a global lower bound.

Figure 3.8 shows how SynchBB, SynchID and Adopt scale up with increasing number of agents on graph coloring problems. The results in Figure 3.8 (left) show that Adopt significantly outperforms both SynchBB and SynchID on graph coloring problems of link density 2. The speed-up of Adopt over SynchBB is 100-fold at 14 agents. The speed-up of Adopt over SynchID is 7-fold at 25 agents and 8-fold at 40 agents. The speedups due to search strategy are significant for this problem class, as exhibited by the difference in scale-up between SynchBB and SynchID. In addition, the figure also show the speedup due exclusively to the asynchrony of the Adopt algorithm. This is exhibited by the difference between SynchID and Adopt, which employ a similar search strategy, but differ in amount of asynchrony. In SynchID, only one agent executes at a time so it has no asynchrony, whereas Adopt exploits asynchrony when possible by allowing agents to choose variable values in parallel. In summary, we conclude that

Adopt is significantly more effective than SynchBB on sparse constraint graphs and the speed-up is due to both its search strategy and its exploitation of asynchronous processing. Adopt is able to find optimal solutions very efficiently for large problems of 40 agents.

Figure 3.8 (right) shows the same experiment as above, but for denser graphs, with link density 3. We see that Adopt still outperforms SynchBB – around 10-fold at 14 agents and at least 18-fold at 18 agents (experiments were terminated after 100000 cycles). The speed-up between Adopt and SynchID, i.e, the speed-up due to concurrency, is 2.06 at 16 agents, 2.22 at 18 agents and 2.37 at 25 agents. Finally, Figure 3.9 shows results from a weighted version of graph coloring where each constraint is randomly assigned a weight between 1 and 10. Cost of solution is measured as the sum of the weights of the violated constraints. We see similiar results on the more general problem with weighted constraints.

Figure 3.10 shows the average total number of messages sent by all the agents per cycle of execution. As the number of agents is increased, the number of messages sent per cycle increases only linearly. This is in contrast to a broadcast mechanism where we would expect an exponential increase. In Adopt, an agent communicates with only neighboring agents and not with all other agents.

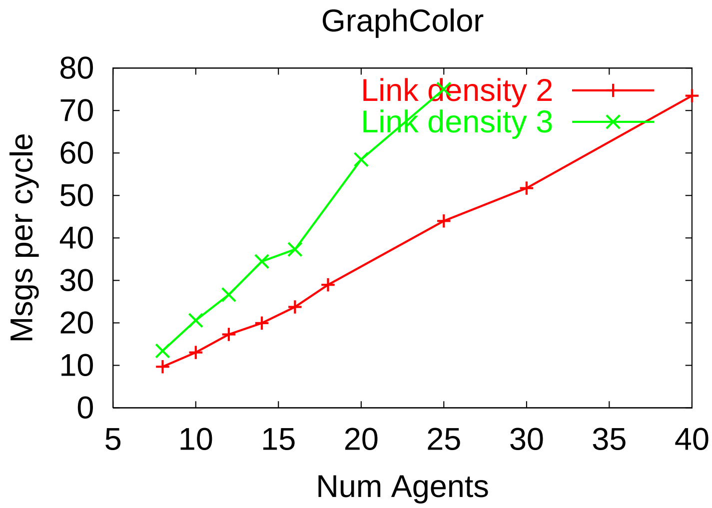


Figure 3.10: Average number of messages per cycle required to find the optimal solution.

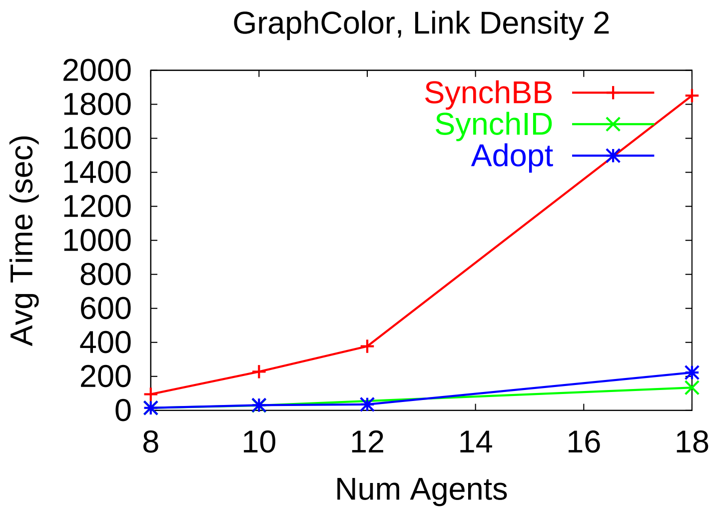


Figure 3.11: Run-time required to find the optimal solution on a single processor (MaxCSP)

.

3.4.1 Measuring Run-Time

In this section, we address a potential criticism of the above experimental results by presenting additional evidence that Adopt, under certain assumptions about communication infrastructure, outperforms the competing algorithms in terms of actual run-time.

While the experimental results presented in the previous section are encouraging, they leave open one important question. In the previous section, we have presented experimental results demonstrating that Adopt requires fewer cycles than the competing algorithms SynchBB and SynchID. This result is important but it is open to the potential criticism that measuring cycles does not measure run-time directly, which is the real evaluation metric of concern. By "run-time" we mean the actual wall clock time from beginning to end of the algorithm execution. The chief concern is that there is no guarantee that the actual run-time of an Adopt cycle (where agents process multiple messages per cycle and do constraint checks every cycle) is equivalent to the run-time of a SynchBB or SynchID cycle (where one agent processes only one message per cycle and may or may not perform any constraint checks in a given cycle). These differences suggest that each cycle in Adopt takes more time than each cycle in SynchBB or SynchID. The question to be answered is: Given that Adopt consumes fewer cycles than the other algorithms but those cycles may take more time,

- *does Adopt really outperform the competing algorithms in terms of actual run-time?*

In this section, we attempt to answer the above question. As mentioned previously, it is problematic to measure the run-time of an asynchronous implementation directly because such results are very sensitive to the particular experimental set-up, underlying communication infrastructure properties, and other variables. These factors are often unpredictable and vary significantly across domains. Therefore we investigate our question in two alternative ways. First, we simply measure the run-time of the algorithm in our single-processor simulated distributed implementation ignoring communication costs and the potential for speedups due to parallelism. The run-time on single-processer puts a parallel algorithm like Adopt is at a significant disadvantage. However, if Adopt outperforms other algorithms under this disadvantage, it provides evidence that Adopt will also outperform them in a truly distributed setting.

Figure 3.11 shows the result of wall-clock cpu-time for executing the algorithms on a single-processor simulated distributed implementation. Even with the disadvantage, Adopt easily outperforms SynchBB. The run-time of Adopt is about equal to SynchID. This is consistent since both algorithms use the same search strategy (lower-bound based search) and differ only in the amount of parallelism. We can guess that when Adopt is executed on a fully distributed system, it will have obtain additional speedups to outperform SynchID.

As a second method to address our question, we use an analytical model to convert synchronous cycles to actual run-time that takes into account both computation time and communication time. In the rest of this section, we present this model and apply it to our experimental results from the previous section. We conclude that when communication time dominates computation time and the communication infrastructure is able to transmit multiple messages in parallel, an asynchronous algorithm that terminates with fewer cycles will be faster in terms of run-time than a synchronous algorithm that consumes greater cycles. In other words, synchronous cycles is a valid metric for estimating run-time under those assumptions. However, if communication latencies are low, i.e., on the same scale as computation, then the comparative computation time per cycle becomes the dominating factor.

We now present an analytical model that allows us to calculate run-time from data collected from synchronous cycle experiments. We first define the following terms:

1) $\mathcal{I}(x_i)$: number of incoming messages processed by x_i per cycle

2) s : computation time to receive one incoming msg

3) $\mathcal{C}(x_i)$: number of constraint checks by x_i per cycle

4) t : computation time to do one constraint check

5) $\mathcal{O}(x_i)$: number of outgoing messages sent by x_i per cycle

6) u : computation time to send one outgoing msg

7) $L(n)$: time required for the communication infrastructure to transmit n msgs simultaneously

Note that the definitions of $\mathcal{I}$, $\mathcal{C}$, and $\mathcal{O}$ have assumed that an agent processes the same number of messages each cycle, performs the same number of constraint checks on each cycle, and sends the same number of messages each cycle (or at least an upper bound on these numbers can be determined). This is true for all the algorithms under consideration. The values of s, t and u are determined by the experimental set-up (efficiency of implementation, programming language, processor speed, operating system, other CPU load on a shared machine, etc..) and may change across different experimental set-ups. The value of L is determined by properties of the underlying communication network. For a given number of messages, we have assumed L is constant (or an average can be determined). Finally, we assume s, t and u are algorithm independent. On the other hand, $\mathcal{I}$, $\mathcal{C}$, and $\mathcal{O}$ are very much dependent on the algorithm but are independent of experimental set-up.

In each cycle, all agents concurrently process all their incoming messages, do constraint checks and send all their outgoing messages. The total computation time for an algorithm whose execution has been measured in terms of synchronous cycles, denoted $total\ cycles$, is given by the following equation.

$$total\ computation\ time = total\ cycles \times \max_{x_i \in Ag}(s \times \mathcal{I}(x_i) + t \times \mathcal{C}(x_i) + u \times \mathcal{O}(x_i)) \tag{3.1}$$

In a sequential algorithm, exactly one agent executes in each cycle and all other agents are idle. The executing agent processes exactly one incoming message, may or may not perform a constraint check, and sends exactly one message. So for a sequential algorithm, we have $\mathcal{I}(x_i) = \mathcal{O}(x_i) = 1$ and $\mathcal{C}(x_i) \geq 0$ for the executing agent x_i, and $\mathcal{I}(x_j) = \mathcal{O}(x_j) = 0$ and $\mathcal{C}(x_j) = 0$ for all other idle agents x_j. Let $\mathcal{C}$ denote the number of constraint checks by the unique agent who executes each cycle. Thus equation (3.1) for a sequential algorithm simplifies to

$$total\ computation\ time \quad = total\ cycles \times (s + t \times \mathcal{C} + u) \tag{3.2}$$

The communication time per cycle is determined by the number of messages transmitted each cycle. The total communication time consumed by an algorithm whose execution has been measured in terms of synchronous cycles is given by the following equation.

$$total\ communication\ time = total\ cycles \times L(\sum_{x_i \in Ag} \mathcal{O}(x_i)) \tag{3.3}$$

Note that we can simplify equation 3.3 for the case of a sequential algorithm where $\sum_{x_i \in Ag} \mathcal{O}(x_i) = 1$.

Finally, the total run-time is given by the sum of total computation time and total communication time over all cycles.

$$\textit{total runtime} = \textit{total computation time} + \textit{total communication time} \tag{3.4}$$

This completes the analytical model. This model will allow us to convert our empirical results from the previous section into calculated run-times. We need to only plug in empirically measured values for the terms 1-7. The rest of this section presents our results.

In our measurements we found the following. The time to process one incoming message is about equal to the time required to send an outgoing message, i.e, $s = u$. Also, this time is about two orders of magnitude slower than the time required to do a constraint check, i.e., $s = u = 100t$. For each algorithm and each problem class, values for $\mathcal{I}$, $\mathcal{C}$, and $\mathcal{O}$ were either empirically measured or a rough upper bound was used.

Next, we make an important assumption on the underlying communication infrastructure. Concurrent algorithms are most suitable for communication infrastructures that allow multiple messages to be transmitted in parallel without significant degradation in overall throughput. For communication networks where this assumption does not hold, i.e. each message must be transferred sequentially, the value of a concurrent algorithm (in terms of efficiency) may be signficantly reduced. Thus, we make the following assumption:

- Assumption: $L(n) = L$, where L is constant for n less than some reasonable number.

For example, agent A1 sending messages to A2 should not degrade communication among two other agents A3 and A4. Examples include communication infrastructure that has multiple parallel channels or the Internet where the two pairs of agents (A1,A2) and (A3,A4) are on different subnets. Radio-frequency communication where (A1,A2) are spatially separated from (A3,A4) also has these properties.

We will calculate run-time for varying values of L. In particular we consider four cases: a) when the time required to communicate per cycle, denoted by L, is on the same order of magnitude as the time required to do a constraint check, denoted by t, i.e., $L = t$, b) when L is one order of magnitude slower, i.e., $L = 10t$, c) when L is two orders of magnitude slower, i.e., $L = 100t$ and finally, d) when L is three orders of magnitude slower, i.e., $L = 1000t$.

Figure 3.12 shows the calculated run-times using the experimental data from Figure 3.8. Graphs are shown for varying values of L. We can see that when communication time is about equal to computation time, the $L = t$ case, Adopt does not outperform SynchID. However as L begins to outweigh t, we see that Adopt begins to do better. In the $L = 1000t$ case, we see that the graph looks similar to Figure 3.8. We can conclude from our analysis that when communication time significantly outweighs computation time, the cycles metric is an accurate substitute for run-time.

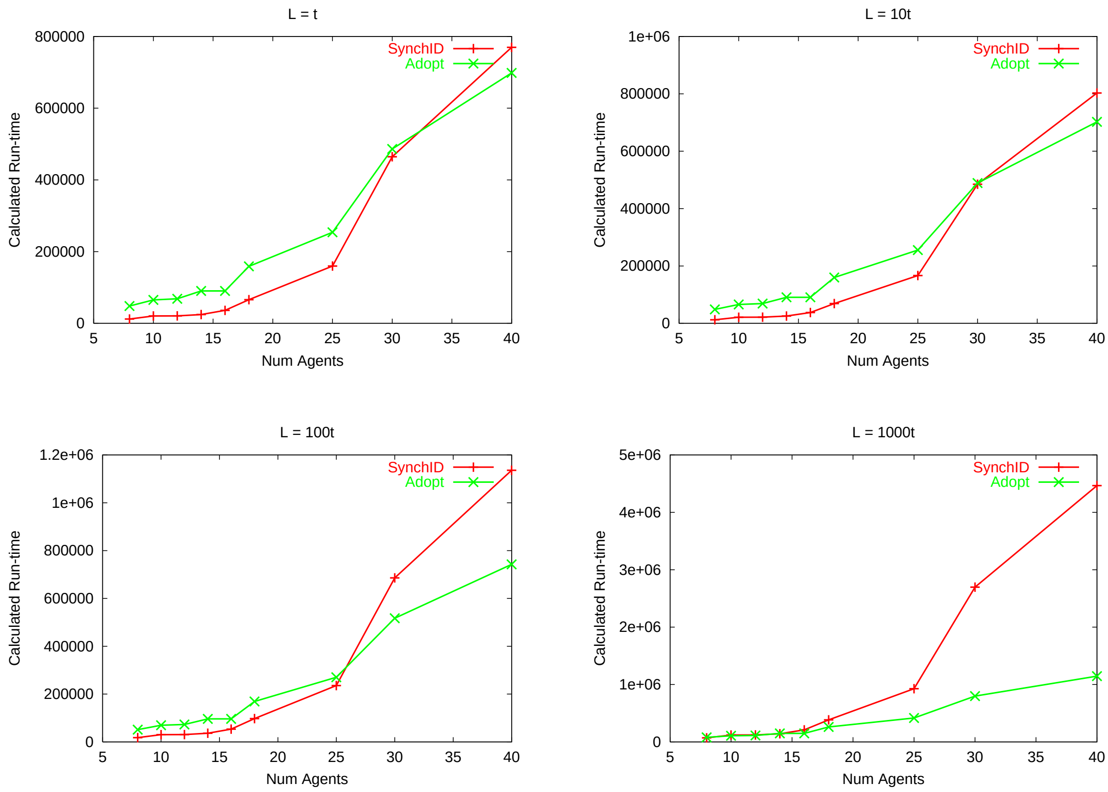


Figure 3.12: Run-time calculated from number of cycles required to find the optimal solution

3.5 Algorithmic Variations for Future Work

Adopt is one example within a space of algorithms that may be designed that exploits our key idea of using lower bounds to perform distributed optimization. In this section, we explore possible algorithmic modifications to Adopt but leave detailed exploration of these issues for future work.

Memory Usage. We consider how Adopt can be modified to obtain efficiency gains at the expense of the polynomial-space bound at each agent. In Adopt, each agent maintains a single $CurrentContext$ as a partial solution and all stored costs are conditioned on the variable values specified in that context. When context changes occur, agents delete all stored costs. This is necessary to maintain the polynomial-space bound. However, in some cases worst-case exponential-space requirements are tolerable either because sufficient memory is available or the worst-case is sufficiently unlikely to occur. In such cases, we may allow agents to store more than one partial solution at a time. Agents should not delete all stored costs when context changes and instead agents should maintain multiple contexts and their associated costs. In this way, if a previously explored context should become current again due to variable value changes at higher agents, then the stored costs will be readily available instead of having to be recomputed. Preliminary experiments (not reported here) have shown this technique can dramatically decrease solution time.

Reducing Communication. We consider how Adopt can be modified to reduce the number of messages communicated. In Adopt, an agent always sends VALUE and COST messages every time it receives a message from another agent, regardless of whether its variable value or costs have changed. As a consequence, an agent often sends a message that is identical to a message that it sent in the immediately prior cycle. Although this is seemingly wasteful, it is a sufficient mechanism to ensure liveness.

However, if other mechanisms are employed to ensure liveness, then it may be possible to reduce the number of messages dramatically[1]. Briefly, an agent can deterimine whether a message should be sent by simply checking whether a message it is about to send is identical to the message it sent in the immediately prior cycle. The message is sent if and only if the message is different from the previous one. Thus, an agent only sends a message if it has new information to communicate.

Sending COST messages to non-parent ancestors. We consider how Adopt can be modified to allow COST messages to be sent to multiple ancestors instead of only to the parent. To see how such reporting may decrease solution time, consider the following scenario. Suppose x_r is the root agent and it has a constraint with neighbor x_i who is very low in the tree, i.e., the length of p is large, where p is the path from x_r to x_i obtained by traversing only parent-child edges in the tree-ordering. If x_r initially chooses a bad variable value that causes a large cost on the constraint shared with x_i, we would like x_r to be informed of this cost as soon as possible so that it may explore other value choices. In Adopt, x_i will send a COST message only to its immediate parent and not to x_r. The parent will then pass the cost up to its parent and so on up the tree. This method of passing costs up the tree is sufficient to ensure completeness, however, the drawback in this case is that since the length of p is large, it will take a

[1]An alternative mechanism for ensuring liveness through the use of timeouts is described in Section 4.3.1.

long time for x_r to be informed of the cost incurred by x_i. Thus, it may take a long time before x_r will abandon its bad choice resulting in wasted search.

To resolve this problem, we may allow an agent to report cost directly to all its neighbors higher in the tree. The key difficulty is that when an agent receives multiple COST messages it cannot safely sum the lower bounds in those messages to compute a new lower bound as in Definition 3 of Section 3.2.1 because double-counting of costs may occur. Such double-counting will violate our completeness guarantee. We can resolve this difficulty by attaching a list of agent names to every COST message (in addition to the information already in the COST messages). This list of names corresponds to those agents whose local costs were used to compute the cost information in that message. A receiving agent can use this list to determine when two COST messages contain overlapping costs.

More precisely, a leaf agent attaches its own name to every COST message it sends. When an agent receives a COST message, it appends its own name to the list contained in that message and attaches the new list to every COST message it sends. When an agent receives multiple COST messages, the lower bounds received in those messages are summed if and only if the attached list of agent names are disjoint. If they are not disjoint, the information in the message with the bigger list is used and the other message is discarded. Although we do not yet have a proof that is method is complete, it seems like a promising approach for improving the Adopt algorithm while maintaining completeness.

Extension to n-ary constraints. Adopt can be easily extended to operate on DCOP where constraints are defined over more than two variables. Suppose we are given a DCOP that contains a ternary constraint $f_{ijk} : D_i \times D_j \times D_k \rightarrow N$ defined over 3 variables x_i, x_j, x_k, as shown in Figure 3.13. The tree ordering procedure must ensure that x_i, x_j and x_k lie on a single path from root to leaf (they may not be in different subtrees since all three are considered neighbors). Suppose x_i and x_j are ancestors of x_k. With binary constraints, the ancestor would send a VALUE message to the descendent. With our ternary constraint, both x_i and x_j will send VALUE messages to x_k. x_k then evaluates the ternary constraint and sends COST messages back up the tree as normal. The way in which the COST message is received and processed by an ancestor is unchanged. Thus, we deal with an n-ary constraint by assigning responsiblity for its evaluation to the lowest agent involved in the constraint. The only difference between evaluation of an n-ary constraint and a binary one is that the lowest agent must wait to receive all ancestors' VALUE messages before evaluating the constraint. For this reason operating on problems with n-ary constraints may decrease concurrency and efficiency of the Adopt algorithm. However this seems unavoidable due to the inherent complexity of n-ary constraints.

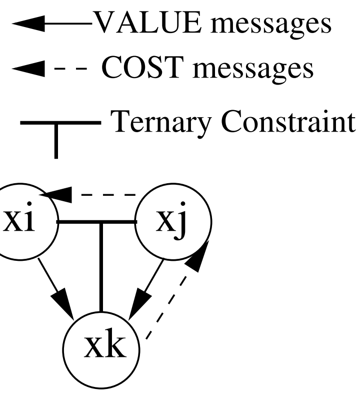


Figure 3.13: A ternary constraint

Extension to multiple variables per agent. Our assumption of one variable per agent can be problematic in domains where agents have complex local subproblems that are more appropriately modeled using multiple variables. We outline two simple methods that exist for dealing with this problem and point out their drawbacks. These methods are also described by Yokoo et al [52] in the context of Distributed Constraint Satisfaction. We then propose a third method which may be more effective than either of these two simple methods.

First, it is always possible to convert a constraint reasoning problem involving multiple variables into a problem with only one variable by defining a new variable whose domain is the cross product of the domains of each of the original variables. This method can in principle be used to convert an agent's complex local subproblem into a subproblem with only one variable. This would allow Adopt to be applied without any modifications. Second, another method is to create multiple virtual agents within a single real agent and assign one local variable to each virtual agent. Each virtual agent then operates as an independent Adopt agent. In principle, this method also allows Adopt to be applied without any modifications.

While the above methods work in principle they may have significant problems in practice. In the first method, the domain size of the new variable will be exponential in the number of variables in the original problem. Exhaustively searching over all the domain values for the new variable can be prohibitively expensive. On the other hand, the virtual agent approach is inefficient since it does not take advantage of the fact that the virtual agents within a single real agent have direct access to each others memory.

A key choice to be made in dealing with multiple variables per agent is whether to form the (DFS tree) ordering over the problem variables or over the agents. The first method described above (in which a local subproblem is converted into a single variable) forms the ordering over the agents. The second method (in which virtual agents are employed) forms the ordering over the problem variables. Forming the ordering over agents is criticised by Yokoo et al. [52] (in the context of DisCSP) because an agent must perform an exhaustive search over its subproblem before it is able to send a NOGOOD message. If a higher agent makes a bad choice, the lower agent must do a lot of work before it can inform the higher agent of the bad choice. Thus, the authors propose an algorithm which chooses to order over problem variables.

However Yokoo's criticism of ordering over agents does not apply in our case of DCOP. The reason is that an agent need not do an exhaustive search over its local subproblem in order to compute a lower bound for its subproblem. Therefore, it can send feedback to higher agents in the form of a lower bound in COST messages without doing exhaustive search. We propose forming the ordering over the agents, but avoid converting the local subproblem into a single variable. Instead, each agent searches over its local subproblem of multiple variables using a centralized lower-bound based optimization method, e.g., IDA* search [33]. IDA* is a suitable method for use in Adopt with multiple variables per agent because it allows an agent to compute a lower bound for its subproblem without exhaustively searching the entire subproblem. In Adopt, an agent could use IDA* to compute a lower bound for its subproblem quickly and send a COST message immediately to its parent without having to exhaustively search its entire subproblem first. A key open question is how the agent should order its local variables before employing IDA*. A rational strategy may be to place higher in the local ordering those variables that have constraints with higher priority agents. In conclusion, we believe this is a promising approach for dealing with multiple variables per agent in Adopt. However, how these approaches may actually work in practice is an empirical question which requires further investigation.

Chapter 4

Limited Time and Unreliable Communication

In this chapter we extend the method presented in the previous chapter to deal with two practical issues that arise in real-world domains: Limited time and unreliable communication. The first section describes our approach for reducing solution time and presents empirical results demonstrating the ability to perform principled tradeoffs between solution time and quality. The second section demonstrates robustness to message loss.

4.1 Limited Time

In this section we consider how agents can perform distributed optimization when sufficient time to find the optimal solution is not available. This is significant because in many time-critical domains there exist hard or soft *deadlines* before which decisions must be made. These deadlines restrict the amount of time available for problem-solving so that it becomes infeasible to determine the optimal solution. This problem is important because it is known that DCOP is NP-hard, so time required to find optimal solutions cannot be done efficiently in the worst-case.

Previous approaches to this problem have typically abandoned systematic global search in favor of incomplete local methods that rely exclusively on local information. This approach is effective in reducing solution time in many domains but the reduction is accomplished by abandoning all theoretical guarantees on solution quality. Instead of theoretical guarantees, empirical evaluations on a necessarily limited number of domains are used to demonstrate algorithm effectiveness. This approach has a three significant drawbacks. First, the reliance on solely empirical evaluation makes it difficult to predict the effectiveness of a given method in new unseen domains. Little can be said about solution quality or solving time when the method is translated to new problems. Second, incomplete local methods cannot guarantee optimal solutions no matter how much time is allowed. Even in situations where more time is available, an incomplete local method is unable to take advantage of this additional time to guarantee better solution quality. Finally, agents cannot know the global quality of the solutions they have obtained. This prevents agents from performing any kind of reasoning about how or whether they should terminate or continue searching for better solutions.

We present a more flexible method called *bounded error approximation* whereby agents can find global solutions that may not be optimal but are within a given distance from optimal. This method decreases solution time for application in time limited domains. Bounded error approximation is similar to incomplete search in that approximate suboptimal solutions are found fast, but is different from incomplete search in that theoretical guarantees on global solution quality are still available.

4.1.1 Bounded-error Approximation

We consider the situation where the user provides Adopt with an error bound b, which is interpreted to mean that any solution whose cost is within b of the optimal is acceptable. More formally, we wish to find any solution S where cost(S) $\leq$ cost(optimal solution) + b. For example in overconstrained graph coloring, if the optimal solution requires violating 3 constraints, $b = 5$ indicates that 8 violated constraints is an acceptable solution. Note that this measure allows a user to specify an error bound without a priori knowledge of the cost of the optimal solution.

The key difficulty is that the cost of the optimal solution is unknown in advance so it is hard to know if the cost of an obtained solution is within the user-defined bound. We solve this problem in Adopt by using the best known lower bound as an *estimate* of the cost of the optimal solution. Then, the agents search for a complete solution whose cost is b over the best known lower bound. If such a solution is found, it can be returned as a solution within the given error bound.

More formally, Adopt can be guaranteed to find a global solution within bound b of the optimal by allowing the root's backtrack threshold to overestimate by b. The root agent uses b to modify its ThresholdInvariant as follows:

- **ThresholdInvariant For Root (Bounded Error):** $min(LB{+}b, UB) = threshold$. The root agent always sets $threshold$ to b over the currently best known lower bound LB, unless the upper bound UB is known to be less than $LB + b$.

Let us revisit the example shown in figure 2.2. We will re-execute the algorithm, but in this case the user has given Adopt an error bound $b = 4$. Instead of initializing $threshold$ to zero, the root agent x_1 will initialize $threshold$ to b. Note that LB is zero upon initialization and UB is Inf upon initialization. Thus, $min(LB + b, UB) = min(4, Inf) = 4$ and the thresholdInvariant above requires x_1 to set $threshold = 4$. In addition, the AllocationInvariant requires x_1 to set $t(0, x_2) = 4$ since the invariant requires that $threshold = 4 = \delta(0) + t(0, x_2) = 0 + t(0, x_2)$ hold.

In figure 4.1.a, all agents again begin by concurrently choosing value 0 for their variable and sending VALUE messages to linked descendents. In addition, x_1 sends a THRESHOLD message to x_2. Upon receipt of this message, x_2 sets $threshold = 4$ (line 8).

Each agent computes LB and UB and sends a COST message to its parent (figure 4.1.b). This was described previously in section 3.2.2 and shown in figure 3.6.b. The execution path is the same here.

Next, x_1 receives x_2's COST message. As before, the received costs will be stored in lines 32-33 as $lb(0, x_2) = 1$ and $ub(0, x_2) = Inf$. In line 37, execution enters the **backTrack** procedure. x_1 computes $LB(1) = \delta(1) + lb(1, x_2) = 0 + 0 = 0$ and $LB(0) = \delta(0) + lb(0, x_2) = 0 + 1 = 1$. Since $LB(1) < LB(0)$, we have $LB = LB(1) = 0$. $UB(0)$ and $UB(1)$ are computed as Inf, so $UB = Inf$. Since $threshold = 4$ is not equal $UB = Inf$, the test in line 38 fails. So far, the execution is exactly as before. Now however, the test in line 40 fails because $LB(d_i) = LB(0) = 1$ is not greater than $threshold = 4$. Thus, x_1 *will not switch value* to $x_1 = 1$ and will instead keep its current value of $x_1 = 0$.

Next, x_2 receives the COST messages sent from x_3 and x_4. The received costs will be stored in lines 32-33 as $lb(0, x_3) = 2$, $ub(0, x_3) = 2$, $lb(0, x_4) = 1$, and $ub(0, x_4) = 1$. In line 37, execution enters the **backTrack** procedure. x_2 computes $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 1 + 2 + 1 = 4$ and $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 2 + 0 + 0 = 2$. Thus, $LB = LB(1) = 2$. Similarly, x_2 computes $UB(0) = \delta(0) + ub(0, x_3) + ub(0, x_4) = 1 + 2 + 1 = 4$ and $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 2 + Inf + Inf = Inf$. Thus, $UB = UB(0) = 4$. Since $threshold = UB = 4$, the test in line 38 succeeds. However, x_2 will not switch value since its current value is the one that minimizes $UB(d)$. Note that the equivalent test in line 45 succeeds, but the test in line 46 fails since x_2 has not yet received a TERMINATE message from x_1. So,

x_2 does not terminate. Instead, execution proceeds to line 52 where a COST message is sent to x_1. This is depicted in figure 4.1.c.

Next, x_1 receives x_2's COST message. The received costs will be stored as $lb(0, x_2) = 2$ and $ub(0, x_2) = 4$. x_1 now computes $LB(1) = \delta(1) + lb(1, x_2) = 0 + 0 + 0$ and $LB(0) = \delta(0) + lb(0, x_2) = 0 + 2 = 2$. Similarly, x_1 computes $UB(1) = \delta(1) + ub(1, x_2) = 0 + Inf = Inf$ and $UB(0) = \delta(0) + ub(0, x_2) = 0 + 4 = 4$. Thus, $UB = UB(0) = 4$. So, now we have the test $threshold == UB$ in line 45 evaluate to true, since $threshold = UB = 4$. Since x_1 is the root, the test in line 47 succeeds and x_1 will terminate with value $x_1 = 0$. It will send a TERMINATE message to x_2 and the other agents will terminate in a similar manner.

In this way, we see the agents have ultimately settled on a configuration with all values equal to 0, with a total cost of 4. Since the optimal solution has cost 0, the obtained solution is indeed within the given error bound of $b = 4$. The solution was found faster because less of the solution space was explored. In particular, note that x_1 never had to explore solutions with $x_1 = 1$.

Theorems 1 and 2 still hold with the bounded-error modification to the Threshold-Invariant. Also, agents still terminate when $threshold$ value is equal UB. The root's final $threshold$ value is the cost of a global solution within the given error bound. Using this error bound, Adopt is able to find a solution faster than if searching for the

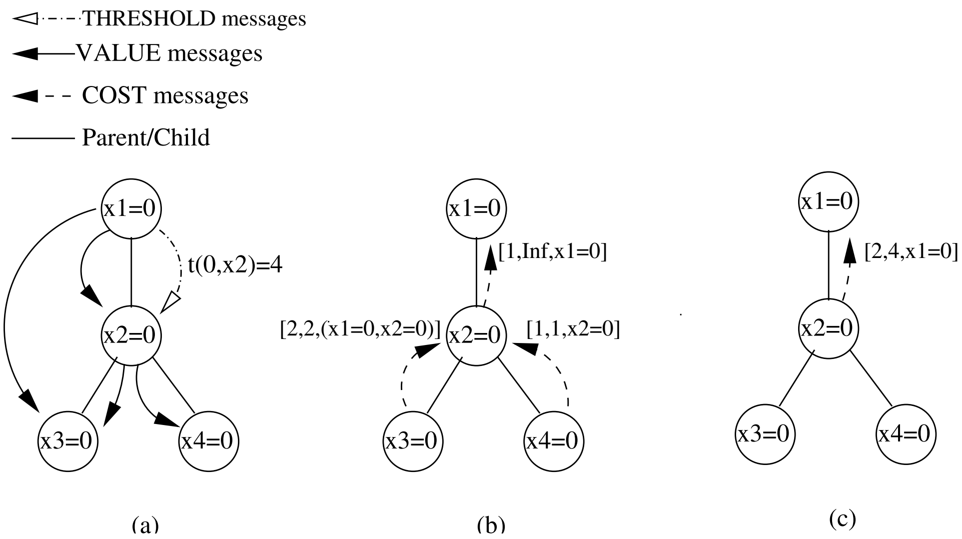


Figure 4.1: Example Adopt execution for the DCOP shown in figure 2.2, with error bound $b = 4$.

optimal solution, thereby providing a method to trade-off computation time for solution quality. This trade-off is principled because a theoretical quality guarantee on the obtained solution is still available.

4.1.2 Experiments

We evaluate the effect on time to solution (as measured by cycles) and the total number of messages exchanged, as a function of error bound b in Figure 4.2. Error bound $b = 0$ indicates a search for the optimal solution. Figure 4.2 (left) shows that increasing the error bound significantly decreases the number of cycles to solution. At 18 agents, Adopt finds a solution that is guaranteed to be within a distance of 5 from the optimal in under 200 cycles, a 30-fold decrease from the number of cycles required to find the optimal solution. Similarly, figure 4.2 (right) shows that the total number of messages exchanged per agent decreases significantly as b is increased.

We evalute the effect on cost of obtained solution as a function of error bound b. Figure 4.3 shows the cost of the obtained solution for the same problems in Figure 4.2. (Data for problems instances of 18 agents is shown, but the results for the other problem instances are similar.) The x-axis shows the "distance from optimal" (cost of obtained solution minus cost of optimal solution for a particular problem instance) and the y-axis shows the percentage of 25 random problem instances where the cost of the obtained solution was at the given distance from optimal. For example, the two bars labeled "$b = 3$" show that when b is set to 3, Adopt finds the optimal solution for 90 percent of the examples and a solution whose cost is at a distance of 1 from the optimal for the remaining 10 percent of the examples. The graph shows that in no cases is the cost of the obtained solution beyond the allowed bound, validating our theoretical results. The graph also shows that the cost of the obtained solutions are often much better than the given bound, in some cases even optimal.

The above results support our claim that varying b is an effective method for doing principled tradeoffs between time-to-solution and quality of obtained solution. These results are significant because, in contrast to incomplete search methods, Adopt provides the ability to find solutions faster when time is limited but without giving up theoretical guarantees on solution quality.

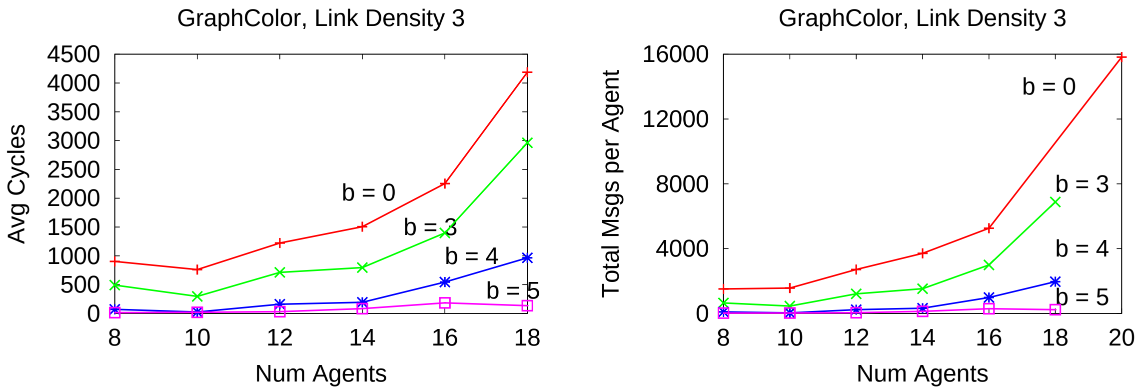


Figure 4.2: Average number of cycles required to find a solution (left) and the average number of messages exchanged per agent (right) for given error bound b.

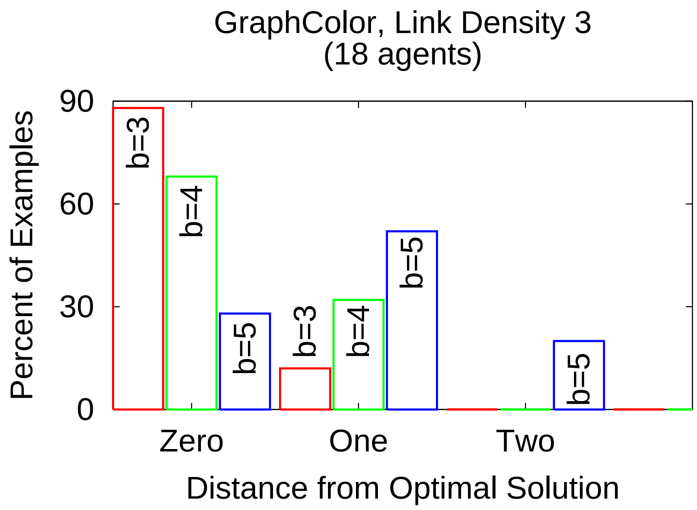


Figure 4.3: Percentage of problem instances where obtained cost was at a given distance from optimal (18 agents)

4.2 Extensions for Future Work

In our view, the bounded-error approximation technique presented above represents a novel method for performing optimization under limited time. Given that the algorithm parameter b provides agents with the ability to effectively trade speed for solution quality, it is natural to begin thinking about ways in which one can use this ability. We believe this opens up a wide array of possiblities for future work in distributed reasoning algorithms. We briefly discuss two interesting avenues.

Anytime Algorithms. When time is limited, agents could potentially set b very high initially to quickly find a greedy solution. After terminating the search, if more time for problem solving is available, they could iteratively decrease the error bound b in attempts to find better solutions. In this way, Adopt can be used in an *anytime* fashion, where at any time during algorithm execution, a candidate solution with some upper bound on cost is available.

Meta-level Reasoning Suppose agents know that they have 20 seconds to find a solution to a given DCOP. How should they set the b parameter to ensure with some sufficient probability that a solution will be found in 20 seconds? One approach is to use meta-level reasoning where agents use performance profiles to predict for a given problem, or class of problems, how long problem-solving is expected to take at a given error bound. Hansen and Zilberstein [15] present some methods for doing this in centralized algorithms. They present a general framework for meta-level control of anytime algorithms [15]. This framework allows a meta-level controller to obtain the highest quality solution when taking into account the time necessary to find it. Their techniques could be adapted to the distributed case using Adopt and bounded-error approximation as tools.

4.3 Unreliable Communication

In this section we consider how agents can perform distributed optimization when message delivery is unreliable, i.e. messages may be dropped. Existing work in DCR algorithms typically assume that communication is perfect. This assumption is problematic because unreliable communication is a common feature of many real-world multiagent domains. Limited bandwidth, interference, loss of line-of-sight are some reasons why communication can fail. We introduce a novel method for dealing with message loss in the context of a particular DCR algorithm named Adopt. The key idea in our approach is to let the DCR algorithm inform the lower error-correction software layer which key messages are important and which can be lost without significant problems. This allows the algorithm to flexibly and robustly deal with message loss. Results show that with a few modifications, Adopt can be guaranteed to terminate with the optimal solution even in the presence of message loss and that time to solution degrades gracefully as message loss probability increases.

In order to provide strong guarantees on the correctness and completeness of DCR algorithms, algorithm designers have typically made the following two assumptions about the communication infrastructure on which the distributed algorithm operates:

- *Reliable*: Delay in delivering a message is finite, i.e., every message sent is eventually received.
- *Atomic*: Order is preserved in transmissions between any pair of agents.

We consider how we can relax the reliability assumption without giving up key algorithm properties such as the guarantee of termination with a correct solution. We assume a simple form of unreliable communication: the communication infrastructure has an unknown *loss probability* $r < 1$, where a message is dropped (not delivered) with probability r. In our experiments we will assume r is constant over time, but this is not strictly necessary. Investigation for relaxation of the atomic assumption is an issue for future work.

A common method for dealing with unreliable channels in communication networks is to implement an error correction layer in software that can ensure reliable message delivery even when the communication infrastructure itself is inherently unreliable. This is typically done through an acknowledgment protocol where ACK messages are used to verify that a message has been received. A number of such protocols have been developed in the field of computer networking, the most popular of which is

TCP [42]. Unfortunately, recent research provides evidence that TCP is infeasible in many types of communication networks important for applications in multiagent systems, such as wireless and ad-hoc networks [2] [30].

Simply relying on a lower layer error-correction mechanism to ensure reliable delivery is an inadequate approach for dealing with unreliable communication infrastructure when developing multiagent algorithms. First, it can significantly increase the number of messages that must be communicated since every message must be acknowledged. Second, a sender cannot send any messages to a given agent x_i until the ACK for a previously sent message is received from x_i for fear of the violating the atomic assumption mentioned earlier. The time cost in waiting for ACKs can degrade performance and reduce the efficiency of the higher-level DCR algorithm. Third, this method is unable to take advantage of the fact that it may be okay for some messages to be lost without large negative effects on the higher-level DCR algorithm.

We propose a novel approach to dealing with message loss in DCR algorithms. We assume the availability of an asynchronous DCR algorithm and a lower error-correction software layer. Instead of relying exclusively on the error-correction layer, we advocate giving the DCR algorithm itself the control to decide which messages must be communicated reliably and which can be lost. The idea is that by requiring only key messages be communicated reliably while allowing other messages to be lost, we can design DCR algorithms that are more flexible and robust to message loss.

We show how this idea can be implemented within the context of the Adopt algorithm and the benefits that are derived. With a few modifications, we show that Adopt is still guaranteed to terminate with the optimal solution even if communication is unreliable. Experimental results show that Adopt's performance degrades gracefully with message loss. We also present results that suggest that artificially introducing message loss even when communication is reliable could be a way to decrease the amount of work agents need to do to find the optimal solution. Indeed, previous work by Fernandez et al. has shown that artificially introducing communication delay in DCR can have beneficial effects on the performance of DCR algorithms.

4.3.1 Algorithm Modifications for Message Loss

Since Adopt is completely asynchronous, we hypothesize it is well suited for operating under unreliable communication infrastructure. In particular, agents are able to process messages no matter when they are received and are insensitive to the order in which messages are received (provided the messages come from different agents). This is in contrast to synchronous algorithms which require messages to be received and sent in a particular order. For example in synchronous algorithms such as Synchronous Branch and Bound [16], if a message is lost no progress can be made until that message is successfully retransmitted.

While asynchrony is a key advantage, the major difficulty that arises is the danger of deadlock. Deadlock can occur for two reasons: the loss of VALUE/COST messages, or the loss of TERMINATE message. We consider each case separately. If VALUE or COST messages are lost, we could have a deadlock situation with agents waiting for each other to communicate. For example, consider two agents x_1 and x_2. x_1 sends a VALUE message to x_2. x_2 evaluates and sends back to x_1 a COST message. Suppose this message gets lost. At this point, x_1 is now waiting for a COST message from x_1, while x_2 is waiting for another VALUE message from x_1. Thus, the loss of one message has resulted in the agents getting deadlocked.

We can overcome deadlock problems arising from loss of VALUE and COST messages due to another key novelty of the Adopt algorithm: Adopt's built-in termination detection. The termination condition allows an agent to locally determine whether the algorithm has terminated. If no messages are received for a certain amount of time and an agent's termination condition is not true, then that agent can conclude that a deadlock may have occurred due to message loss. The agent can then resend VALUE and COST messages to its neighbors in order to trigger the other agents and break the deadlock. This method requires the implementation of a timeout mechanism at each agent. The value of the timeout can be set according to the average time between successive messages. This solution is more flexible and efficient than the alternative approach of dealing with message loss at a lower error-correction software layer. Instead, the algorithm intelligently determines when messages need to be resent by using the local termination condition as a guide. This method is also stateless in the sense that an agent does not need to remember the last message it sent in case a resend is needed. The agent can simply send out its current value and current cost whenever a timeout occurs.

Another reason deadlock can occur is if a TERMINATE message is lost. Agents terminate execution in response to the TERMINATE message received from their parents. If that message gets lost, an agent has no knowledge that the rest of the agents have come to a stop. The TERMINATE messages are essentially a distributed snapshot mechanism [6] whereby the agents determine that the termination condition is true at all the other agents. Unfortunately, distributed snapshot algorithms require reliable communication. Thus, Adopt requires that TERMINATE messages be sent reliably. This is can be done through an acknowledgment protocol where each TERMINATE message sent must be acknowledged by the recipient by responding to the sender with an ACK message. The sender will resend its message if an ACK is not received after certain amount of time. The sender continues to resend until an ACK is eventually received. Since $r < 1$, the TERMINATE message and the ACK will eventually (in the limit) go through. Since TERMINATE messages only need to be communicated once between parent and child, the overhead for this is not very severe, and is certainly less than requiring every message to be communicated reliably.

With these modifications it can be ensured that Adopt will eventually terminate with the optimal solution, regardless of the amount of network disturbance, so long as the probability of a message being lost is less than 1. To see that Adopt will eventually terminate, realize the deadlock detection timeout will ensure that an agent x_i will not sit waiting forever for a message that may not come when its own termination condition is not true. Instead, x_i will continue sending messages to its children until a reply is received. Thus, each child will eventually report to x_i a lower bound that is also an upper bound. When this occurs, x_i's termination condition will finally be true and it can terminate.

4.3.2 Experiments

We experiment on distributed graph 3-coloring.[1] One node is assigned to one agent who is responsible for its color. Global cost of solution is measured by the total number of violated constraints. We experiment with graphs of link density 3 – a graph with link density d has dn links, where n is the number of nodes in the graph. We average over 6 randomly generated problems for each problem size and each problem was run three times for each loss probability, for a total of 18 runs for each datapoint. Each agent runs in a separate thread and time to solution is measured as CPU time. We use a uniform

[1] I thank Syed Muhammed Ali and Rishi Goel for their assistance with these experiments

timeout value of 10 seconds. We ensured consistent system load between runs and each run produced an optimal solution.

Table 4.1 shows the relative change in running time as a percentage of the running time when communication is perfect ($r = 0$). We see that as loss probability r increases from 0% to 10%, the running time increases very little – only 5.88% for 10 agents and 4.66% for 12 agents. At loss probablity of 20%, we begin to see more severe effects on running time – 20.95% for 10 agents and 19.31% for 12 agents. The data provides initial evidence that Adopt's performance degrades gracefully as message loss rate increases.

In addition to solution time, we would also like to know if the agents are doing more or less work when messages are being lost as compared to when communication is perfect. One measure of "work" is the total number of messages processed. Table 4.2 shows the relative change in the total number of messages processed as a percentage of the number of messages processed when communication is perfect. We see that agents process fewer messages as message loss rate increases – around 8% less for 12 agents at 20% loss. These results show that agents are able to obtain a solution of optimal quality but by processing fewer messages as compared to the perfect communication case. This suggests that artificially introducing message loss even when communication is reliable could be a way to decrease the amount of work agents need to do to find the optimal solution. In fact, recent work by Fernandez et al. has shown that artificially introducing

Table 4.1: Running time as a percentage of when there is zero loss.

Loss rate(r)	8 Agents	10 Agents	12 Agents
0%	100.00%	100.00%	100.00%
2%	99.61%	98.84%	100.17%
5%	103.94%	100.40%	100.01%
10%	110.78%	105.88%	104.66%
20%	128.93%	120.95%	119.31%

Table 4.2: Number of messages processed (rcvd) as a percentage of when there is zero loss

Loss rate(r)	8 Agents	10 Agents	12 Agents
0%	100.00%	100.00%	100.00%
2%	98.80%	98.01%	98.36%
5%	98.44%	96.47%	95.27%
10%	98.63%	95.04%	93.02%
20%	97.58%	92.24%	92.59%

communication delay in DCR can have beneficial effects on the performance of DCR algorithms [11]. This is something we will explore in future work.

To summarize, we found that while sending acknowledgements for every message was excessive and too expensive, and sending none was problematic, the right tradeoff was to allow the DCR algorithm to control which key messages need to be sent reliably. We showed that this method allows an asynchronous algorithm to tolerate message loss and still terminate with the globally optimal solution. Empirical results showed that time-to-solution increased gradually as message loss rate is increased which is a desireable property in a DCR algorithm. We also found that agents need to process fewer messages to find the optimal solution when messages may be lost, which suggests that an active loss mechanism may improve algorithm performance.

Chapter 5

Modeling Real-world Problems

In this chapter, we illustrate that the Distributed Constraint Reasoning (DCR) paradigm can be used to represent and solve an important class of Distributed Resource Allocation problem. We develop an abstract formalization of the Distributed Resource Allocation problem that allows detailed complexity analysis and general mappings into the DCR representation. Previous work in modeling Distributed Resource Allocation has lacked a systematic formalization of the problem and a general solution strategy. In particular, formalizations that allow detailed complexity analysis and general mappings into DCR are missing.

Figure 5.1 depicts the overall methodology. First, we propose an abstract formalization of Distributed Resource Allocation (shown as box (a)) that is able to capture both the distributed and dynamic nature of the problem. The abstract formalization is significant because it allows us to understand the complexity of different types of

Distributed Resource Allocation problems and allows tractable problem subclasses to be identified. Next, we develop generalized mappings into DCR (shown as box (b)). In particular, we will consider two specific DCR representations: DCOP which was previously defined in chapter 2 and DyDisCSP which is defined in this chapter. DCOP can represent optimization problems but cannot represent dynamic problems. On the other hand, DyDisCSP is able to represent dynamic problems at the expense of limiting to a satisfaction-based representation. The Adopt algorithm can be used to solve DCOPs while the LD-AWC algorithm presented in this chapter can be used to solve DyDisCSPs. Unfortunately sound and complete algorithms for dynamic optimization are currently unavailable. Thus, we will present mappings into both DCOP and DyDisCSP, each concerned with different aspects of the problem (optimization and dynamics).

The general mapping strategies are significant because they enable existing DCR technologies to be brought to bear directly onto the distributed resource allocation problem. In particular with these mappings, we can use DCR algorithms like Adopt and LD-AWC in particular to automatically solve distributed resource allocation problems. In addition, they allow future algorithmic advances in DCR to also be directly applied to the distributed resource allocation problem without significant re-modeling effort. Thus, our formalism and generalized mappings may provide researchers with tools for both representing and solving their resource allocation problem using DCR.

In section 5.1, we will describe the details of the distributed sensor network problem as a concrete example of the Distributed Resource Allocation problem. This domain is then used to illustrate the formal definitions in section 5.2 which formalize our abstract model of Distributed Resource Allocation. Properties and complexity classes are defined in section 5.3 and 5.4. In section 5.5, we define the Dynamic Distributed Constraint Satisfaction Problem (DyDisCSP) and the LD-AWC algorithm for solving DyDisCSP. Sections 5.6 and 5.7 define our mappings of Distributed Resource Allocation into DyDisCSP. Section 5.8 defines our mapping into DCOP.

5.1 Application Domain

Our distributed sensor network problem introduced previously is used to illustrate the difficulties described above and to also illustrate our formalization of Distributed Resource Allocation that is described later. The domain consists of multiple stationary sensors, each controlled by an independent agent, and targets moving through their sensing range. Figure 2.1 shows the hardware. Each sensor is equipped with a Doppler radar with three sector heads. Each sector head covers 120 degrees and only one sector can be active at a time. While all of the sensor agents must choose to activate their sector heads to track the targets, there are some key difficulties in such tracking.

The first difficulty is that the domain is inherently distributed. In order for a target to be tracked accurately, at least three agents must collaborate. They must concurrently

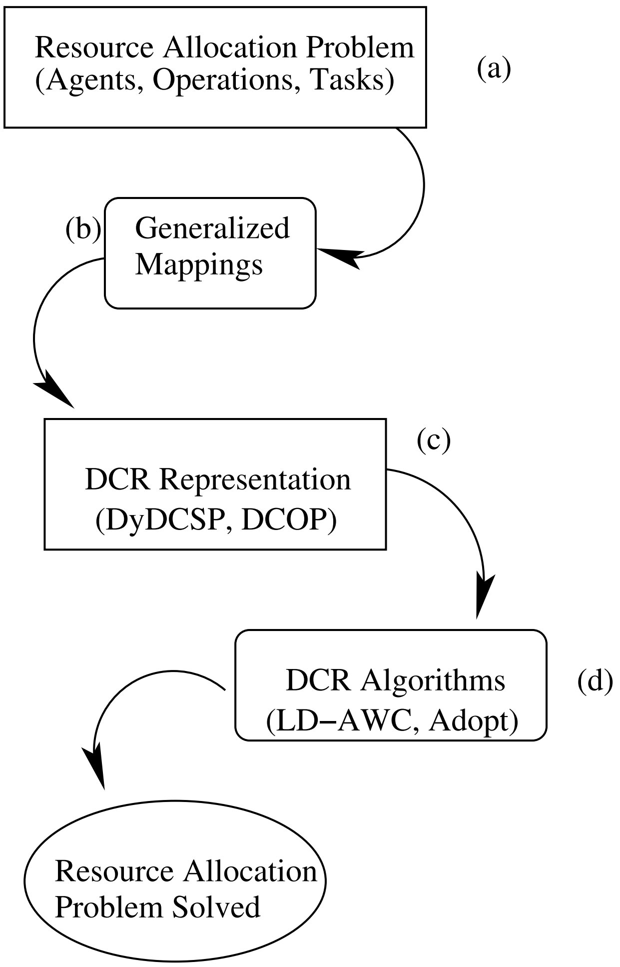


Figure 5.1: Graphical depiction of the described methodology.

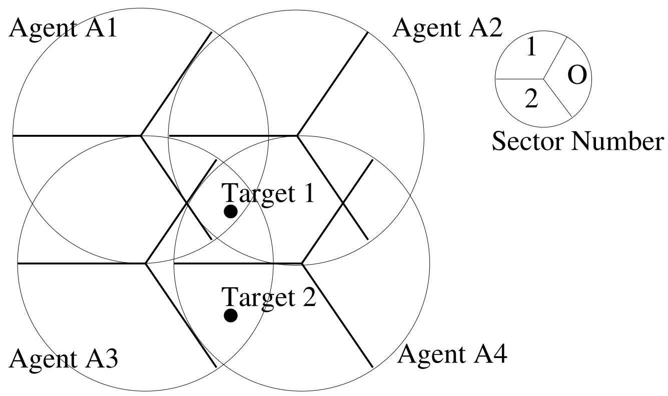


Figure 5.2: A schematic of four sensor nodes.

activate their sectors so the target is sensed by at least three overlapping sectors. For example, in Figure 5.2, if an agent A1 detects target 1 in its sector 0, it must inform two of its neighboring agents, A2 and A4 for example, so that they activate their respective sectors that overlap with A1's sector 0.

The second difficulty with accurate tracking is that when an agent is informed about a target, it may face ambiguity in which sector to activate. Each sensor can detect only the distance and speed of a target, so an agent that detects a target can only inform another agent about the general area of where a target may be, but cannot tell other agents specifically which sector they must activate. For example, suppose there is only target 1 in Figure 5.2 and agent A1 detects that a target is present in its sector 0. A1 can tell A2 that a target is somewhere in the region of its sector 0, but it cannot tell A2 which sector to activate because A2 has two sectors (sector 1 and 2) that overlap with A1's sector 0. In order to resolve this ambiguity, A2 may be forced to first activate its sector 1, detect no target, then try its sector 2. Thus, activating the correct sector requires a collaboration between A1 and A2. A1 informs A2 that a target exists in some ambiguous region and A2 then resolves the remaining ambiguity. The significance here is that no single agent can determine the correct allocation of all sectors to targets.

The third difficulty is that resource contention may occur when multiple targets must be tracked simultaneously. For instance, in Figure 5.2, A4 needs to decide whether to track target 1 or target 2 and it cannot do both since it may activate only one sector at a time. A4 should choose to track target 2 since there is no other way for target 2 to be tracked. A4 is "critical" for tracking target 2. In general, determining whether an agent is "critical" for a particular target requires non-local information about targets out of an agent's immediate sensing range. In this example, note that target 1 and 2 are tasks that conflict with one another. Targets that are spatially distant do not conflict with each other and thus can easily be tracked without resource contention. Thus, as we will see, the relationship among tasks will affect the difficulty of the overall resource allocation problem.

Finally, the situation is dynamic because targets move through the sensing range. Even after agents find a configuration that is accurately tracking all targets, they may have to reconfigure themselves as targets move over time.

The above application illustrates the difficulty of resource allocation among distributed agents in a dynamic environment. Lack of a formalism for dynamic distributed resource allocation problem can lead to ad-hoc methods which cannot be easily reused. Instead, our adoption of a formal model allows our problem and its solution to be stated in a more general way, possibly increasing our solution's usefulness. More importantly, a formal treatment of the problem also allows us to study its complexity and provide other researchers with some insights into the difficulty of their own resource allocation problems. Finally, a formal model allows us to provide guarantees of soundness and completeness of our results. The next section presents our formal model of resource allocation.

5.2 Formal Definitions

A Distributed Resource Allocation Problem consists of 1) a set of agents that can each perform some set of operations, and 2) a set of tasks to be completed. In order to be completed, a task requires some subset of agents to perform certain operations. We can define a task by the operations that agents must perform in order to complete it. The problem is to find an allocation of agents to tasks such that all tasks are performed. In this way, we view the agents as the resources to be allocated. This problem is formalized next.

- **Definition 1**: A Distributed Resource Allocation Problem is a structure $<\mathcal{A}g$, Ω, $\Theta>$ where

- $\mathcal{A}g$ = $\{A_1, A_2, ..., A_n\}$ is a set of agents.
- Ω = $\{O^1_1, O^1_2, ..., O^i_p, ..., O^n_q\}$ is a set of operations, where operation O^i_p denotes the p'th operation of agent A_i. An operation can either succeed or fail. Let $Op(A_i)$ denote the set of operations of A_i.
- $\Theta = \{T_1, T_2, ..., T_n\}$ is a set of tasks, where each task $T \in \Theta$ is a set of sets $\{t_1, t_2, ..., t_n\}$ and each $t_i \in T$ is a set of operations. Each t_i is called a *minimal set*.

Intuitively, the minimal sets of a task specify the alternative ways (sets of operations) to perform the task. We assume that tasks are either performed or not performed. We do not model varying degrees of task performance. Tasks are performed by executing all the operations in one of its minimal sets. No more (or less) operations are needed. Thus, we require that each minimal set t_r of a given task T is "minimal" in the sense that no other minimal set in T is a subset of t_r. Beyond the minimal resources required for a task, we do not model more complex task features such as task duration, task difficulty, task performance cost, etc. although we note that these are features that are important in many domains and deserve attention.

We assume operations in $Op(A_i)$ are *mutually exclusive* in the sense that an agent can only perform one operation at a time. This is an assumption that holds in our domain of distributed sensor networks and in many other domains. However, in order to capture domains where agents are able to perform multiple operations and do many tasks at once, this assumption must be relaxed in future work.

A *solution* to a resource allocation problem is to choose a minimal set for each present task such that the chosen minimal sets do not conflict. We say that two minimal sets of two different tasks *conflict* if they each contain an operation belonging to the same agent. Since we assume that an agent can only perform one operation at a time, it is not possible for all the operations in two conflicting minimal sets to be executed simultaneously. In general when there are too many tasks and not enough agents, it may not be possible to choose non-conflicting minimal sets for every present task. In such cases, we wish the agents to allocate resources only to the most important tasks. More formally, we wish to find $\Theta_{sat} \subseteq \Theta_{current}$, such that $<\mathcal{A}g$, Ω, $\Theta_{sat}>$ has a solution and $\mid \Theta_{sat} \mid$ is maximized. In other words, we must solve the optimization problem where we wish to choose a subset of the tasks so that the maximum number of tasks are completed.

To illustrate this formalism in the distributed sensor network domain, we cast each sensor as an agent and activating one of its (three) sectors as an operation. We will use O^i_p to denote the operation of agent A_i activating sector p. For example, in Figure 5.2, we have four agents, so $\mathcal{A}g$ = $\{A_1, A_2, A_3, A_4\}$. Each agent can perform one of three operations, so Ω = $\{O^1_0$,O^1_1, O^1_2,O^2_0,O^2_1, O^2_2, O^3_0,O^3_1, O^3_2, O^4_0,O^4_1, O^4_2 $\}$. To specify the subset of operations belonging to a particular agent, say A_1, we use $Op(A_1)$ = { O_0^1,O_1^1, O_2^1 }.

We distinguish between tasks that are *present* and *not present*. Present tasks require resources, while tasks that are not present do not require resources. Tasks change from being present to not present and vice versa over time. Θ (defined above) corresponds to the set of all tasks, present or not. We use $\Theta_{current}$ ($\subseteq \Theta$) to denote the set of tasks that are currently present. We call a resource allocation problem *static* if $\Theta_{current}$ is constant over time and *dynamic* otherwise. In our distributed sensor network example, since targets come and go, the problem is a dynamic one. We do not model resource allocation problems where the resources may be dynamic (i.e., where agents may come and go).

Returning to our example, we now define our task set Θ. We define a separate task for each region of overlap of sectors where a target may potentially be present. In other words, tasks correspond to spatially segmented regions of space. The existence of an actual target in a particular segmented region corresponds to a present task. Regions of overlap that do not currently contain a target are tasks that do not currently need to be performed. Associated with each region is the set of operations that can sense that space, i.e., the sensor sectors that cover it. In the situation illustrated in Figure 5.2, we have two targets shown. We define our current task set as $\Theta_{current}$ = $\{T_1, T_2\}$. In the figure, there are many other regions of overlap for which tasks are defined, but we omit description of the full set Θ for simplicity. Task T_1 requires any three of the four possible agents to activate their corresponding sector, so we define a minimal set corresponding to all the $\binom{4}{3}$ combinations. Thus, T_1 = $\{\{O_0^1, O_2^2, O_0^3\}, \{O_2^2, O_0^3, O_1^4\}, \{O_0^1, O_0^3, O_1^4\}, \{O_0^1, O_2^2, O_1^4\}\}$. Note that the subscript of the operation denotes the number of the sector the agent must activate. In the example, task T_2 can only be tracked by two sectors, so T_2 = $\{\{O_0^3, O_2^4\}\}$.

For each task, we use $\Upsilon(T_r)$ to denote the union over all the minimal sets of T_r, and for each operation, we use $T(O_p^i)$ to denote the set of tasks T_r for which O_p^i can contribute, that is, those tasks that include O_p^i in $\Upsilon(T_r)$. For instance, $\Upsilon(T_1)$ = $\{O_0^1, O_2^2, O_0^3, O_1^4\}$ and $T(O_0^3)$ = $\{ T_1, T_2\}$ in the example above.

The set $\Theta_{current}$ is determined by the environment and not necessarily immediately known to all the agents. It also changes over time. A key difficulty is how the agents can come to know which tasks are currently present. We will discuss this question shortly after the following two definitions. We assume that when an agent *executes* one of its operations, the operation either succeeds or fails depending on the presence or absence of tasks in the environment at the time the operation is executed.

- **Definition 2**: Let $O_p^i \in \Omega$ be an operation executed by A_i. If $\exists\, T_r \in \Theta_{current}$ such that $O_p^i \in \Upsilon(T_r)$, then O_p^i *succeeds*. If O_p^i has no corresponding task in $\Theta_{current}$, the operation *fails*.

In our example, if agent A_1 executes operation O_0^1 (activates sector 1) and if $T_1 \in \Theta_{current}$ (target 1 is present), then O_0^1 will succeed (A_1 will detect a target), otherwise it will fail. Note that our notion of operation failure corresponds to a sensor signal indicating the absence of a task, not actual hardware failure. Hardware failure or sensor noise is an issue not modeled. However, an actual system built using this formalism has been able to incorporate techniques for dealing with noise and failure by using a two-layered architecture, where a lower layer of the implementation deals with these issues[36].

We say a task is (being) *performed* when all the operations in some minimal set succeed. More formally,

- **Definition 3**: $\forall T_r \in \Theta$, T_r is *performed* iff there exists a minimal set $t_r \in T_r$ such that all the operations in t_r succeed. A task that is not present cannot be performed, or equivalently, a task that is performed must be included in $\Theta_{current}$.

The intuition is that as long as the task is present, it can (and should) be performed. When it is no longer present, it cannot (and need not) be performed. This is different from the notion of agents working on a task until it is "completed". In our formalism, agents have no control over task duration.

For example, task T_2 is performed (target T_2 is tracked) if A_3 executes operation O_0^3 and $A4$ executes operation O_2^4. The task continues to be performed as long as those operations are being executed and the task is present i.e., the target remains in the specified region of space.

To summarize our formalism and terminology:

- Tasks are *present* or *not present*, as determined by the environment.
- Operations are *executed* by the agents and executed operations *succeed* or *fail* depending on the presence or absence of corresponding tasks.
- Tasks are *performed* when all the operations in some minimal set succeed.

As mentioned above, a key difficulty is how the agents can come to know which tasks are currently present. In our model, agents execute their operations to not only perform existing tasks, but also to detect when new tasks have appeared and when existing tasks disappear. Thus, agents must continually interleave problem solving and operator execution.

If a new task appears and an agent executes its operation, the operation will succeed and signal to the agent that some task is present. It may not necessarily know exactly which task is present, since there may be multiple tasks for which the same operation may succeed. Aside from this difficulty (which we will address in our solution methodology), another tricky issue is ensuring that every new task will eventually be detected by some agent, i.e., some agent will execute its operation to detect it. We must avoid situations where all agents are busy doing other tasks or sleeping and ignoring new tasks.

This can be done in various ways depending on the particular domain. In the sensor domain, we require agents to "scan" for targets by activating different sectors when they are currently not involved in tracking any target. Thus, our model relies on the following assumption which ensures that no present task goes unnoticed by everyone.

- **Notification assumption**:
  - (i) If task T_r is present, then at least one agent executes an operation for T_r: $\forall T_r \in \Theta$, if $T_r \in \Theta_{current}$, then $\exists\, O_p^i \in \Upsilon(T_r)$ such that O_p^i is executed (and since T_r is present, O_p^i succeeds).
  - ii) $\forall T_s (\neq T_r) \in \Theta_{current}$, $O_p^i \notin \Upsilon(T_s)$.

(ii) states that the notifying operation O_p^i (from (i)) must not be part of any other present task. This only implies that the success of operation O_p^i will uniquely identify the task T_r among all present tasks, but not necessarily among all (e.g., not present) tasks. Thus, the difficulty of global task ambiguity remains. This assumption is only needed to prevent two present tasks from being detected by one agent through the execution of a single operation, in which case the agent must choose one of them, leaving the other task undetected by anyone. In distributed sensor networks, hardware restrictions preclude the possibility of two targets being detected by a single sector, so this assumption is naturally satisfied.

This concludes our model. In later sections, we will map this Distributed Resource Allocation model into a Dynamic Distributed Constraint Satisfaction Problem. We will show how these mappings and associated algorithms can be used to address the problems of distribution, task dynamics, resource contention, and global task ambiguity that arise within our model.

5.3 Properties of Resource Allocation

We now state some definitions that will allow us to categorize a given resource allocation problem and analyze its difficulty. In particular, we notice some properties of task and inter-task relationships. We choose to identify these properties because, as we will see, they have a bearing on the computation complexity of the overall resource allocation problem. Definitions 4 through 7 are used to describe the complexity of a given task in a given problem, i.e., the definitions relate to properties of an *individual task*. Next, definitions 8 through 10 are used to describe the complexity of inter-task relationships, i.e., the definitions relate to the interactions between a *set of tasks*.

5.3.1 Task Complexity

For our purposes, we consider a particular notion of task complexity, namely, the expressiveness allowed by the minimal set representation. In its most general form (the

Unrestricted class defined below), one can express a variety of minimal sets. However, for certain problem classes, we can limit the minimial set representation to reduce computational complexity. We now define some of these types of problem classes.

One class of resource allocation problems have the property that each task requires any k agents from a pool of n ($n \geq k$) available agents. That is, the task contains a minimal set for each of the $\binom{n}{k}$ combinations. The following definition formalizes this notion.

- **Definition 4**: $\forall\, T_r \in \Theta$, T_r is **task-$\binom{n}{k}$-exact** iff T_r has exactly $\binom{n}{k_r}$ minimal sets of size k_r, where n = $\mid \Upsilon(T_r) \mid$ and $k_r (\leq n)$ depends on T_r.

For example, the task T_1 (corresponding to target 1 in Figure 5.2) is task-$\binom{4}{3}$-exact because it has exactly $\binom{4}{3}$ minimal sets of size $k = 3$, where $n = 4 = \mid \Upsilon(T_1) \mid$. The following definition defines the class of resource allocation problems where every task is task-$\binom{n}{k}$-exact.

- **Definition 5** : $\binom{n}{k}$**-exact** denotes the class of resource allocation problems $< \mathcal{A}g$, Ω, $\Theta >$ such that $\forall\, T_r \in \Theta$, T_r is task-$\binom{n}{k_r}$-exact.

We find it useful to define a special case of $\binom{n}{k}$-exact resource allocation problems, namely those when $k = n$. Intuitively, all agents are required so each task contains only a single minimal set.

- **Definition 6**: $\binom{n}{n}$**-exact** denotes the class of resource allocation problems $<\mathcal{A}g$, Ω, $\Theta>$ such that $\forall\, T_r \in \Theta$, T_r is task-$\binom{n_r}{k_r}$-exact, where $n_r = k_r =\mid \Upsilon(T_r) \mid$.

For example, the task T_2 (corresponding to target 2 in Figure 5.2) is task-$\binom{2}{2}$-exact.

- **Definition 7**: **Unrestricted** denotes the class of resource allocation problems $<\mathcal{A}g$, Ω, $\Theta>$ with no restrictions on tasks.

Note that $\binom{n}{n}$-exact $\subset$ $\binom{n}{k}$-exact $\subset$ Unrestricted.

5.3.2 Task Relationship Complexity

The following definitions refer to relations between tasks. We define two types of *conflict-free* to denote resource allocation problems that have solutions, or equivalently, problems where all tasks can be performed concurrently.

- **Definition 8**: A resource allocation problem is called **Strongly Conflict Free (SCF)** if for all T_r, $T_s \in \Theta_{current}$ and $\forall\ A_i \in \mathcal{A}g$, $\mid Op(A_i) \cap \Upsilon(T_r) \mid + \mid Op(A_i) \cap \Upsilon(T_s) \mid \leq 1$, i.e., no two tasks have in common an operation from the same agent.

The SCF condition implies that we can choose any minimal set out of the given alternatives for a task and be guaranteed that it will lead to a solution where all tasks are performed, i.e., no backtracking is ever required to find a solution.

- **Definition 9**: A resource allocation problem is called **Weakly Conflict Free (WCF)** if there exists some choice of minimal set for every present task such that all the chosen minimal sets are non-conflicting.

The WCF condition is much weaker that the SCF condition since it only requires that there exists some solution. However, a significant amount of search may be required to find it. Finally, we define problems that may not have any solution.

- **Definition 10**: A resource allocation problem that cannot be assumed to be WCF is called **(possibly) over-constrained (OC)**. In OC problems, all tasks may not necessarily be able to be performed concurrently because resources are insufficient.

Note that SCF $\subset$ WCF $\subset$ OC.

5.4 Complexity Classes of Resource Allocation

Given the above properties, we can define 9 subclasses of problems according to their task complexity and inter-task relationship complexity: SCF and $\binom{n}{n}$-exact, SCF and $\binom{n}{k}$-exact, SCF and unrestricted, WCF and $\binom{n}{n}$-exact, WCF and $\binom{n}{k}$-exact, WCF and unrestricted, OC and $\binom{n}{n}$-exact, OC and $\binom{n}{k}$-exact, OC and unrestricted.

Table 5.1 summarizes our complexity results for the subclasses of resource allocation problems just defined. The columns of the table, from top to bottom, represent increasingly complex tasks. The rows of the table, from left to right, represent increasingly complex inter-task relationships. We refer the reader to [29] for detailed proofs.

Although our formalism and mappings addresses dynamic problems, our complexity analysis here deals with a static problem. A dynamic resource allocation problem can be cast as solving a sequence of static problems, so a dynamic problem is at least as hard as a static one. Furthermore, all our complexity results are based on a centralized problem solver. In terms of computational complexity, a distributed problem can always be solved by centralizing all the information. However, we note that this model of complexity ignores issues such as communication costs, communication delays, message loss, limited communication range/bandwith, etc.

Theorem 4 *Unrestricted SCF resource allocation problems can be solved in time linear in the number of tasks*

Proof: Greedily choose any minimal set for each task. They are guaranteed not to conflict by the Strongly Conflict Free condition. □

Theorem 5 $\binom{n}{n}$*-exact WCF resource allocation problems can be solved in time linear in the number of tasks.*

Proof: Greedily choose the single minimal set for each task.

Theorem 6 $\binom{n}{k}$*-exact WCF resource allocation problems can be solved in time polynomial in the number of tasks and operations.*

Proof: We can convert a given $\binom{n}{k}$-exact resource allocation problem to a network-flow problem, which is known to be polynomial time solvable [31]. We first construct a tripartite graph from a given resource allocation problem and then convert the graph into a network-flow problem.

Let $<\mathcal{A}g, \Omega, \Theta>$ be an $\binom{n}{k}$-exact problem. Construct a tripartite graph as follows:

- Create three empty sets of vertices, U, V, and W and an empty edge set E.
- For each task $T_r \in \Theta$, add a vertex u_r to U.
- For each agent $A_i \in \mathcal{A}g$, add a vertex v_i to V.
- For each agent A_i, for each operation $O_p^i \in Op(A_i)$, add a vertex w_p^i to W.
- For each agent A_i, for each operation $O_p^i \in Op(A_i)$, add an edge between vertices v_i, w_p^i to E.
- For each task T_r, for each operation $O_p^i \in \Upsilon(T_r)$, add an edge between vertices u_r, w_p^i to E.

We convert this tripartite graph into a network-flow graph in the following way. Add two new vertices, a supersource s, and supersink t. Connect s to all vertices in V and assign a max-flow of 1. For all edges among V, W, and U, assign a max-flow of 1. Now, connect t to all vertices in U and for each edge (u_r, t), assign a max-flow of k_r. We now have a network flow graph with an upper limit on flow of $\sum_{i=1}^{|\theta|} k_i$.

We show that the resource allocation problem has a solution if and only if the max-flow is equal to $\sum_{i=1}^{|\theta|} k_i$.

$\Rightarrow$ Let a solution to the resource allocation problem be given. We will now construct a flow equal to $\sum_{i=1}^{|\theta|} k_i$. This means, for each edge between vertex u_r in U and t, we must assign a flow of k_r. It is required that the in-flow to u_r equal k_r. Since each edge between W and U has capacity 1, we must choose k_r vertices from W that have an edge into u_r and fill them to capacity. Let T_r be the task corresponding to vertex u_r, and $t_r \in T_r$ be the minimal set chosen in the given solution. We will assign a flow of 1 to all edges (w_p^i, u_r) such that w_p^i corresponds to the operation O_p^i that is required in t_r. There are exactly k_r of these. Furthermore, since no operation is required for two different tasks, when we assign flows through vertices in U, we will never choose w_p^i again. For vertex w_p^i such that the edge (w_p^i, u_r) is filled to its capacity, assign a flow of 1 to the edge (v_i, w_p^i). Here, when a flow is assigned through a vertex w_p^i, no other flow is assigned through $w_q^i \in Op(A_i)$ ($p \neq q$) because all operations in $Op(A_i)$ are mutually exclusive. Therefore, v_i's outflow cannot be greater than 1. Finally, the assignment of flows from s to V is straightforward. Thus, we will always have a valid flow (inflow=outflow). Since all edges from U to t are filled to capacity, the max-flow is equal to $\sum_{i=1}^{|\theta|} k_i$.

$\Leftarrow$ Assume we have a max-flow equal to $\sum_{i=1}^{|\theta|} k_i$. Then for each vertex u_r in U, there are k_r incoming edges filled to capacity 1. By construction, the set of vertices in

W matched to u_r corresponds to a minimal set in T_r. We choose this minimal set for the solution to the resource allocation problem. For each such edge (w_p^i, u_r), w_p^i has an in-capacity of 1, so every other edge out of w_p^i must be empty. That is, no operation is required by multiple tasks. Furthermore, since outgoing flow thorough v_i is 1, no more than one operation in $Op(A_i)$ is required. Therefore, we will not have any conflicts between minimal sets in our solution. □

Theorem 7 *Determining whether an unrestricted resource allocation problem is Weakly Conflict Free is NP-Complete.*

Proof: We reduce from 3 coloring problem. For reduction, let an arbitrary instance of 3-color with colors c_1, c_2, c_3, vertices V and edges E, be given. We construct the resource allocation problem as follows:

- For each vertex $v \in V$, add a task T_v to Θ.
- For each task $T_v \in \Theta$, for each color c_k, add a minimal set $t_v^{c_k}$ to T_v.
- For each edge $v_i, v_j \in E$, for each color c_k, add an operator $O_{v_i,v_j}^{c_k}$ to Ω and add this operator to minimal sets $t_{v_i}^{c_k}$ and $t_{v_j}^{c_k}$.
- Assign each operator to a unique agent $A_{O_{v_i,v_j}^{c_k}}$ in $\mathcal{A}g$.

Figure 5.3 illustrates the mapping from a 3 node graph to a resource allocation problem. With the mapping above, we can show that the 3-color problem has a solution if and only if the constructed resource allocation problem is weakly conflict free.

$\Rightarrow$ Assume the 3-color problem has a solution. Then there exists a coloring such that no adjacent vertices have the same color. Let v_i and v_j be two arbitrary vertices that are colored c_k and c_l. Then we can choose $t_{v_i}^{c_k}$ and $t_{v_j}^{c_l}$ as minimal sets for tasks T_{v_i} and T_{v_j}, respectively. We need to show that the WCF condition holds between these two tasks for all agents. We will show it for one agent and the proof for all agents is identical. Let $O_{v_i,v_m}^{c_k}$ be an operator in $t_{v_i}^{c_k}$. By construction, $\{ O_{v_i,v_m}^{c_k} \}$ = Op($A_{O_{v_i,v_m}^{c_k}}$). So we have $\mid t_{v_i}^{c_k} \cap Op(A_{O_{v_i,v_m}^{c_k}}) \mid$ = 1. Then, the WCF condition is violated just in case $O_{v_i,v_m}^{c_k}$ is in $t_{v_j}^{c_l}$. This is only possible if m=j and k=l. But m=j means that the two vertices we picked are adjacent and k=l means they are colored the same color, which cannot be a solution to the 3-color problem. So the problematic case is ruled out and the resource allocation problem is WCF.

$\Leftarrow$ Assume our constructed resource allocation problem is WCF. Let $t_{v_i}^{c_k}$ and $t_{v_j}^{c_l}$ be WCF minimal sets for tasks T_{v_i} and T_{v_j}. Then we can color v_i and v_j with colors c_k and c_l respectively. The only case where there could be a problem is if (v_i, v_j) is in E and k=l. Assume this is the case. Then, by construction, there exists an operator $O_{v_i,v_j}^{c_k}$ that is in $t_{v_i}^{c_k}$. But this operator is also in $t_{v_j}^{c_k}$ $(= t_{v_j}^{c_l})$, which violates WCF. So the problematic case is ruled out and we have a valid 3-coloring. $\square$

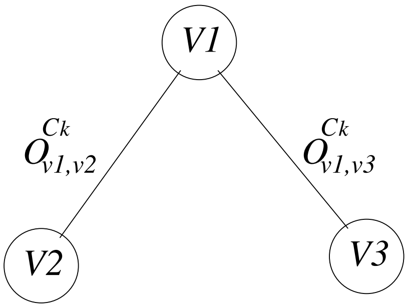
In OC problems sufficient resources may not be available to complete all present tasks. Instead, we may wish to find $\Theta_{sat} \subseteq \Theta_{current}$, such that $<\mathcal{A}g$, Ω, $\Theta_{sat}>$ has a solution and $\mid \Theta_{sat} \mid$ is maximized. In other words, we wish to choose a subset of the tasks so that the maximum number of tasks are completed. We show complexity of this problem to be NP-Complete.

Theorem 8 *The OC Distributed Resource Allocation problem is NP-Complete.*

Proof: We show that a special case is NP-Complete, namely, assume the problem is $\binom{n}{n}$-exact. If we are given a subset of Θ, we can determine if the RAP is solvable in linear time (by Theorem 5). So the problem is in NP. To show this problem is NP-hard, we will reduce from the INDEPENDENT-SET problem [31]. INDEPENDENT-SET is defined as: Let $G = (V, E)$ be an undirected graph, and let $I \subseteq V$. The set I is *independent* if whenever $i, j \in I$ then there is no edge between i and j. The goal is to find the largest independent set in graph G.

The reduction from INDEPENDENT-SET is as follows. For each node i, we create a task T_i with exactly one minimal set. For each edge, $i, j \in E$, we create an operation $O_{i,j}$ and add it to the minimal set of T_i and T_j. Finally, create one agent for each operation. We can see that two tasks conflict if and only if their nodes in G have an edge between them. Let $I \subseteq V$ be is a solution to the INDEPENDENT-SET. Let Θ_{sat} be the set of tasks corresponding to the nodes in I. There is no edge between any $i, j \in I$, so there cannot be any operation in common between T_i and T_j in Θ_{sat}. Thus, Θ_{sat} is the solution to the resource allocation problem. The reverse direction is similar.

□



Color = {R, G, B}

$T_{V1} = \{\{O^R_{v1,v2}, O^R_{v1,v3}\}, \{O^G_{v1,v2}, O^G_{v1,v3}\}, \{O^B_{v1,v2}, O^B_{v1,v3}\}\}$

$T_{V2} = \{\{O^R_{v1,v2}\}, \{O^G_{v1,v2}\}, \{O^B_{v1,v2}\}\}$

$T_{V3} = \{\{O^R_{v1,v3}\}, \{O^G_{v1,v3}\}, \{O^B_{v1,v3}\}\}$

Figure 5.3: Reduction of graph 3-coloring to Resource Allocation Problems

Table 5.1: Complexity Classes of Resource Allocation, n = size of task set Θ, m = size of operation set Ω. Columns represent task complexity and rows represent inter-task relationship complexity.

	SCF	WCF	OC
$\binom{n}{n}$-exact	O(n)	O(n)	NP-Complete
$\binom{n}{k}$-exact	O(n)	O($(n+m)^3$)	NP-Complete
unrestricted	O(n)	NP-Complete	NP-Complete

5.5 Dynamic Distributed CSP (DyDisCSP)

In order to solve resource allocation problems captured by our formalized model, we will use distributed constraint satisfaction techniques. The following section defines the notion of a Dynamic Distributed Constraint Satisfaction Problem (DyDisCSP). Existing approaches to distributed constraint satisfaction fall short for our purposes because they cannot capture the dynamic aspects of the problem. In dynamic problems, a solution to the resource allocation problem at one time may become obsolete when the underlying tasks have changed. This means that once a solution is obtained, the agents must continuously monitor it for changes and must have a way to express such changes in the problem. This section presents DyDisCSP in order to address this shortcoming, DisCSP assumes that the set of constraints are fixed in advance. This assumption is problematic when we attempt to apply DisCSP to domains where the environment is unknown and changes over time. For example, in distributed sensor networks, agents do not know where the targets will appear and how they will move. This makes it difficult to specify the DisCSP constraints in advance. Rather, we desire agents to sense the environment and then activate or deactivate constraints depending on the result of the sensing action. We formalize this idea next.

We take the definition of DisCSP one step further by defining Dynamic DCSP (DyDisCSP). A DyDisCSP is a DisCSP where constraints are allowed to be dynamic, i.e., agents are able to add or remove constraints from the problem according to changes in the environment. More formally,

- **Definition 11**: A *dynamic* constraint is given by a tuple (P, C), where P is a arbitrary predicate that is evaluated to true or false by an agent sensing its environment and C is a familiar constraint from DisCSP.

When P is true, C must be satisfied in any DyDisCSP solution. When P is false, it is okay for C to be violated. An important consequence of dynamic DisCSP is that agents no longer terminate when they reach a stable state. They must continue to monitor P, waiting to see if it changes. If its value changes, they may be required to search for a new solution. Note that a solution when P is true is also a solution when P is false, so the deletion of a constraint does not require any extra computation. However, the converse does not hold. When a constraint is added to the problem, agents may be forced to compute a new solution. In this work, we only need to address a restricted form of DyDisCSP where only *local constraints* are allowed to be dynamic. We will see that this is sufficient to model the types of problems we are interested in. Next, we discuss how we can solve such restricted DyDisCSPs through a simple modification to an existing DisCSP algorithm.

Asynchronous Weak Commitment (AWC) [52] is a sound and complete algorithm for solving DisCSPs. An agent with local variable A_i, chooses a value v_i for A_i and sends this value to agents with whom it has external constraints. It then waits for and responds to messages. When the agent receives a variable value ($A_j = v_j$) from another agent, this value is stored in an AgentView. Therefore, an AgentView is a set of pairs $\{(A_j, v_j), (A_k, v_k), ...\}$. Intuitively, the AgentView stores the current value of non-local variables. A subset of an AgentView is a "NoGood" if an agent cannot find a value for its local variable that satisfies all constraints. For example, an agent with variable A_i may find that the set $\{(A_j, v_j), (A_k, v_k)\}$ is a NoGood because, given these values for A_j and A_k, it cannot find a value for A_i that satisfies all of its constraints. This means that these value assignments cannot be part of any solution. In this case, the agent will request that the others change their variable value and a search for a solution continues. To guarantee completeness, a discovered NoGood is stored so that that assignment is not considered in the future.

The most straightforward way to attempt to deal with dynamism in DisCSP is to consider AWC as a subroutine that is invoked anew everytime a constraint is added. Unfortunately, in domains such as ours, where the problem is dynamic but does not change drastically, starting from scratch may be prohibitively inefficient. Another option, and the one that we adopt, is for agents to continue their computation even as local constraints change asynchronously. The potential problem with this approach is that when constraints are removed, a stored NoGood may now become part of a solution. We solve this problem by requiring agents to store their own variable values as part of non-empty NoGoods. For example, if an agent with variable A_i finds that a value v_i does not satisfy all constraints given the AgentView $\{(A_j, v_j)$, $(A_k,\ v_k)\}$, it will store the set $\{(A_i, v_i)$, (A_j, v_j), $(A_k,\ v_k)\}$ as a NoGood. With this modification to AWC, NoGoods remain "no good" even as local constraints change. Let us call this modified algorithm Locally-Dynamic AWC (LD-AWC) and the modified NoGoods "LD-NoGoods" in order to distinguish them from the original AWC NoGoods. The following lemma establishes the soundness and completeness of LD-AWC.

Lemma I: LD-AWC is sound and complete.

The soundness of LD-AWC follows from the soundness of AWC. The completeness of AWC is guaranteed by the recording of NoGoods. A NoGood logically represents a set of assignments that leads to a contradiction. We need to show that this invariant is maintained in LD-NoGoods. An LD-NoGood is a superset of some non-empty AWC

NoGood and since every superset of an AWC NoGood is no good, the invariant is true when a LD-NoGood is first recorded. The only problem that remains is the possibility that an LD-NoGood may later become good due to the dynamism of local constraints. A LD-NoGood contains a specific value of the local variable that is no good but never contains a local variable exclusively. Therefore, it logically holds information about external constraints only. Since external constraints are not allowed to be dynamic in LD-AWC, LD-NoGoods remain valid even in the face of dynamic local constraints. Thus the completeness of LD-AWC is guaranteed.

5.6 Mapping SCF Problems into DyDisCSP

We now describe a solution to the SCF subclass of resource allocation problems, defined in Definition 8 of Section 5.2, by mapping onto DyDisCSP. We choose DyDisCSP instead of DCOP for two reasons: DyDisCSP is able to represent dynamic problems and the SCF condition guarantees that a satisfactory solution exists so optimization is not necessary. Our goal is to provide a general mapping, named Mapping I, that allows any dynamic unrestricted SCF resource allocation problem to be modeled as DyDisCSP by applying this mapping.

Mapping I is motivated by the following idea. The goal in DyDisCSP is for agents to choose values for their variables so all constraints are satisfied. Similarly, the goal in resource allocation is for the agents to choose operations so all tasks are performed.

Therefore, in our first attempt we map agents to variables and operations of agents to values of variables. For example, if an agent A_i has three operations it can perform, $\{O^i_1, O^i_2, O^i_3\}$, then the variable corresponding to this agent will have three values in its domain. However, this simple mapping attempt fails due to the dynamic nature of the problem; operations of an agent may not always succeed. Therefore, we define two values for every operation, one for success and the other for failure. In our example, this would result in six values for each variable A_i: $\{O^i_1$yes, O^i_2yes, O^i_3yes, O^i_1no, O^i_2no, O^i_3no $\}$.

It turns out that even this mapping is inadequate due to ambiguity. Ambiguity arises when an operation can be required for multiple tasks but only one task is actually present. To resolve ambiguity, we desire agents to be able to not only communicate about which operation to perform, but also to communicate for which task they intend the operation. For example in Figure 5.2, Agent A3 is required to activate the same sector for both targets 1 and 2. We want A3 to be able to distinguish between the two targets when it communicates with A2, so that A2 will be able to activate its correct respective sector. For each of the values defined so far, we will define new values corresponding to each task that an operation may serve.

Mapping I: Given a Resource Allocation Problem $\langle \mathcal{A}g, \Omega, \Theta \rangle$, the corresponding DyDisCSP is defined over a set of n variables.

- A = $\{A_1, A_2,..., A_n\}$, one variable for each $A_i \in$ Ag. We will use the notation A_i to interchangeably refer to an agent or its variable.

The domain of each variable is given by:

- $\forall A_i \in \mathcal{A}g$, Dom($A_i$) = $\bigcup\limits_{O_p^i \in \Omega} O_p^ixT(O_p^i)x\{yes,no\}$.

In this way, we have a value for every combination of operations an agent can perform, a task for which this operation is required, and whether the operation succeeds or fails. For example in Figure 5.2, Agent A3 has one operation (sector 0) with two possible tasks (target 1 and 2). Although the figure does not show targets in sector 1 and sector 2 of agent A3, let us assume that targets may appear there for this example. Thus, let task T_3 be defined as a target in A3's sector 1 and let task T_4 be defined as a target in A3's sector 2. This means A3 would have 8 values in its domain: $\{O_0^3T_1$yes, $O_0^3T_1$no, $O_0^3T_2$yes, $O_0^3T_2$no, $O_1^3T_3$yes, $O_1^3T_3$no, $O_2^3T_4$yes, $O_2^3T_4$no $\}$.

A word about notation: $\forall$ $O_p^i \in \Omega$, the set of values in O_p^ix$T(O_p^i)$x$\{$yes$\}$ will be abbreviated by the term O_p^i*yes and the assignment A_i = O_p^i*yes denotes that $\exists v \in$ O_p^i*yes such that $A_i = v$. Intuitively, the notation is used when an agent detects that an operation is succeeding, but it is not known which task is being performed. This is analogous to the situation in the distributed sensor network domain where an agent may detect a target in a sector, but does not know its exact location. Finally, when a variable A_i is assigned a value, the corresponding agent executes the corresponding operation.

Next, we must constrain agents to assign "yes" values to variables only when an operation has succeeded. However, in dynamic problems, an operation may succeed at some time and fail at another time since tasks are dynamically added and removed from the current set of tasks to be performed. Thus, every variable is constrained by the following *dynamic* local constraints (as defined in Section 5.5).

- **Dynamic Local Constraint 1 (LC1)**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$,

  LC1(A_i) = (P, C), where Predicate P: O_p^i succeeds.

  Constraint C: $A_i = O_p^i$*yes

- **Dynamic Local Constraint 2 (LC2)**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$,

  LC2(A_i) = (P, C), where Predicate P: O_p^i does not succeed.

  Constraint C: $A_i \neq O_p^i$*yes

The truth value of P is not known in advance. Agents must execute their operations, and based on the result, locally determine if C needs to be satisfied. In dynamic problems, where the set of current tasks is changing over time, the truth value of P will also change over time, and hence the corresponding DyDisCSP will need to be continually monitored and resolved as necessary.

We now define the External Constraint (EC) between variables of two different agents. EC is a normal static constraint and must always be satisfied.

- **External Constraint**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$, $\forall A_j \in A$, EC(A_i, A_j): (1) A_i = $O_p^i T_r$yes, and

(2) $\forall t_r \in T_r$, $O_p^i \in t_r$, $\exists q\ O_q^j \in t_r$.

$\Rightarrow A_j$ = $O_q^j T_r$yes

The EC constraint requires some explanation. It says that if A_i detects a task, then other agents in minimal set t_r must also help with the task. In particular, Condition (1) states that an agent A_i is executing a successful operation O_p^i for task T_r. Condition (2) quantifies the other agents whose operations are also required for T_r. If A_j is one of those agents, i.e., O_q^j is an operation that can help perform T_r, the consequent requires A_j to choose operation O_q^j. Note that every pair of variables A_i and A_j have an EC constraint between them. If A_j is not required for T_r, condition (2) is false and EC is trivially satisfied.

5.6.1 Correctness of Mapping I

We now show that Mapping I can be used to model a given SCF resource allocation problem as a DyDisCSP. Theorem 9 states that our DyDisCSP always has a solution. This means the constraints as defined above are not inconsistent and thus, it is always possible to solve the resulting DyDisCSP. Theorem 10 then states that if agents reach a solution, all tasks are (being) performed. Note that the converse of the Theorem 10 does not hold, i.e. it is possible for agents to be performing all tasks *before* a solution to the DyDisCSP is reached. This is due to the fact that when all current tasks are being performed, agents whose operations are not necessary for the current tasks could still be violating some constraints.

Theorem 9 *Given an unrestricted SCF Resource Allocation Problem $\langle \mathcal{A}g,\Omega,\Theta\rangle$, $\Theta_{current}$ $\subseteq \Theta$, a solution always exists for the DyDisCSP obtained from Mapping I.*

Proof: We proceed by presenting a solution to any given DyDisCSP problem obtained from Mapping I.

Let $B = \{A_i \in A \mid \exists T_r \in \Theta_{current}, \exists O_p^i \in \Upsilon(T_r)\}$. B contains precisely those agents who have an operation that can contribute to some current task. We will first assign values to variables in B, then assign values to variables that are not in B. If $A_i \in B$, we assign $A_i = O_p^i T_r yes$, where $T_r \in \Theta_{current}$ and $O_p^i \in \Upsilon(T_r)$. We know such T_r and O_p^i exist by the definition of B. If $A_i \notin B$, we may choose any $O_p^i T_r no \in$ Domain(A_i) and assign $A_i = O_p^i T_r no$.

To show that this assignment is a solution, we first show that it satisfies the EC constraint. We arbitrarily choose two variables, A_i and A_j, and show that EC(A_i, A_j) is satisfied. We proceed by cases. Let $A_i, A_j \in A$ be given.

- *case 1:* $A_i \notin B$ Since $A_i = O_p^i T_r no$, condition (1) of EC constraint is false and thus EC is trivially satisfied.

- *case 2:* $A_i \in B, A_j \notin B$ $A_i = O_p^i T_r yes$ in our solution. Let $t_r \in T_r$, $O_p^i \in t_r$. We know that $T_r \in \Theta_{current}$ and since $A_j \notin B$, we conclude that $\nexists O_q^j \in t_r$. Condition (2) of the EC constraint is false and thus EC is trivially satisfied.

- *case 3:* $A_i \in B, A_j \in B$ $A_i = O_p^i T_r yes$ and $A_j = O_q^j T_s yes$ in our solution. Let $t_r \in T_r$, $O_p^i \in t_r$. T_s and T_r must be strongly conflict free since both are in $\Theta_{current}$. If $T_s \neq T_r$, then $\nexists\ O_n^j \in \Omega$, $O_n^j \in t_r$. Condition (2) of EC(A_i,A_j) is false and thus EC is trivially satisfied. If $T_s = T_r$, then EC is satisfied since A_j is helping A_i perform T_r.

Next, we show that our assignment satisfies the LC constraints. If $A_i \in B$ then $A_i = O_p^i T_r yes$, and LC1, regardless of the truth value of P, is clearly not violated. Furthermore, it is the case that O_p^i succeeds, since T_r is present. Then the predicate P of LC2 is not true and thus LC2 is not present. If $A_i \notin B$ and $A_i = O_p^i T_r no$, it is the case that O_p^i is executed and, by definition, does not succeed. Then, the predicate P of LC1 is not satisfied and thus LC1 is not present. LC2, regardless of the truth value of P, is clearly not violated. Thus, the LC constraints are satisfied by all variables. We can conclude that all constraints are satisfied and our value assignment is a solution to the DyDisCSP. □

Theorem 10 *Given an unrestricted SCF Resource Allocation Problem* $\langle \mathcal{A}g,\Omega,\Theta \rangle$, $\Theta_{current} \subseteq \Theta$ *and the DyDisCSP obtained from Mapping I, if an assignment of values to variables in the DyDisCSP is a solution, then all tasks in* $\Theta_{current}$ *are performed.*

Proof: Let a solution to the DyDisCSP be given. We want to show that all tasks in $\Theta_{current}$ are performed. We proceed by choosing a task $T_r \in \Theta_{current}$. Since our choice is arbitrary and tasks are strongly conflict free, if we can show that it is indeed performed, we can conclude that all members of $\Theta_{current}$ are performed.

Let $T_r \in \Theta_{current}$ be given. By the **Notification Assumption**, some operation O_p^i, required by T_r will be executed. However, the corresponding agent A_i, will be unsure as to which task it is performing when O_p^i succeeds. This is due to the fact that O_p^i may be required for many different tasks. It may choose a task, $T_s \in T(O_p^i)$, and LC1 requires it to assign the value $O_p^i T_s$yes. We will show that A_i could not have chosen incorrectly since we are in a solution state. The EC constraint will then require that all other agents A_j, whose operations are required for T_s also execute those operations and assign $A_j = O_q^j T_s yes$. We are in a solution state, so LC2 cannot be present for A_j. Thus, O_q^j succeeds. Since all operations required for T_s succeed, T_s is performed. By definition, $T_s \in \Theta_{current}$. But since we already know that T_s and T_r have an operation in common, the Strongly Conflict Free condition requires that $T_s = T_r$. Therefore, T_r is indeed performed.□

5.7 Mapping WCF Problems into DyDisCSP

This section begins with a discussion of the difficulty in using Mapping I for solving WCF problems. This leads to the introduction of a second mapping, Mapping II, which is able to map WCF problems into DyDisCSP.

Our first mapping has allowed us to solve SCF resource allocation problems. However, when we attempt to solve WCF resource allocation problems with this mapping, it fails because the DyDisCSP becomes overconstrained. This is due to the fact that Mapping I requires all agents who can possibly help perform a task to do so. If only three out of four agents are required for a task, Mapping I will still require all four agents to perform the task. In some sense, this results in an overallocation of resources to some tasks. This is not a problem when all tasks are independent as in the SCF case. However, in the WCF case, this overallocation may leave other tasks without sufficient resources to be performed. One way to solve this problem is to modify the constraints in the mapping to allow agents to reason about relationships among tasks. However, this requires adding n-ary ($n > 2$) external constraints to the mapping. This is problematic in a distributed situation because there are no efficient algorithms for non-binary distributed CSPs. Existing methods require extraordinary amounts of inter-agent communication. Instead, we create a new mapping by extending mapping I to n-ary constraints, then taking its dual representation. In the dual representation, variables correspond to tasks and values correspond to operations. This allows all n-ary constraints to be *local* within an agent and all external constraints are reduced to equality constraints. Restricting n-ary constraints to be local rather than external is more efficient because it reduces the amount of communication needed between agents. This new mapping, Mapping II, allocates only minimal resources to each task, allowing WCF problems to be solved. Mapping II is described next and proven correct. Here, each agent has a variable for each task in which its operations are included.

Mapping II: Given a Resource Allocation Problem $\langle \mathcal{A}g, \Omega, \Theta \rangle$, the corresponding DyDisCSP is defined as follows:

- **Variables:** $\forall T_r \in \Theta, \forall O_p^i \in \Upsilon(T_r)$, create a DyDisCSP variable $T_{r,i}$ and assign it to agent A_i.

- **Domain:** For each variable $T_{r,i}$, create a value $t_{r,i}$ for each minimal set in T_r, plus a "NP" value (not present). The NP value allows agents to avoid assigning resources to tasks that are not present and thus do not need to be performed.

In this way, we have a variable for each task and a copy of each such variable is assigned to each agent that has an operation for that task. For example in Figure 5.2, Agent A1 has one variable, $T_{1,1}$, Agent A2 has one variable $T_{1,2}$, Agent A3 has two variables, $T_{1,3}$ and $T_{2,3}$, one for each task it can perform, and Agent A4 has two variables, $T_{1,4}$ and $T_{2,4}$. The domain of each $T_{1,i}$ variable has five values, one for each of the four minimal sets as described in Section 5.2, plus the NP value.

Next, we must constrain agents to assign non-NP values to variables only when an operation has succeeded, which indicates the presence of the corresponding task. However, in dynamic problems, an operation may succeed at some time and fail at another time since tasks are dynamically added and removed from the current set of tasks to be performed. Thus, every variable is constrained by the following dynamic local constraints.

- **Dynamic Local (Non-Binary) Constraint (LC1)**:

  $\forall A_i \in \mathcal{A}g$, $\forall O_p^i \in Op(A_i)$, let B = $\{\ T_{r,i} \mid O_p^i \in T_r\ \}$. Then let the constraint be defined as a non-binary constraint over the variables in B as follows:

  Predicate P: O_p^i succeeds

  Constraint C: $\exists T_{r,i} \in$ B $T_{r,i} \neq$ NP.

- **Dynamic Local Constraint (LC2)**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$, let the constraint be defined on $T_{r,i}$ as follows:

  Predicate P: O_p^i does not succeed

  Constraint C: $T_{r,i}$ = NP.

We now define the constraint that defines a valid allocation of resources and the external constraints that require agents to agree on a particular allocation.

- **Static Local Constraint (LC3)**: $\forall T_{r,i}, T_{s,i}$, if $T_{r,i} = t_{r,i}$ and $T_{s,i} = t_{s,i}$, then $t_{r,i}$ and $t_{s,i}$ cannot conflict. NP does not conflict with any value.

- **External Constraint (EC)**: $\forall i, j, r \; T_{r,i} = T_{r,j}$.

For example, if Agent A4 assigns $T_{1,4} = \{O_0^1, O_2^2, O_2^4\}$, then LC3 says it cannot assign a minimal set to its other variable $T_{2,4}$, that contains any operation of either Agent A1, A2 or A4. Since $T_{2,4}$ has only one minimal set, $\{O_0^3, O_2^4\}$ which contains Agent A4, the only compatible value is NP. Note that if Target 1 and 2 are both present simultaneously as shown in Figure 5.2, the situation is overconstrained since the NP value will be prohibited by LC1.

5.7.1 Correctness of Mapping II

We will now prove that Mapping II can be used to represent any given WCF Resource Allocation Problem as a DyDisCSP. As in Mapping I, the Theorem 11 shows that our DyDisCSP always has a solution, and the Theorem 12 shows that if agents reach a solution, all current tasks are performed.

Theorem 11 *Given a WCF Resource Allocation Problem $\langle \mathcal{A}g,\Omega,\Theta \rangle$, $\Theta_{current} \subseteq \Theta$, there exists a solution to DyDisCSP obtained from Mapping II.*

Proof: For all variables corresponding to tasks that are not present, we can assign the value "NP". This value satisfies all constraints except possibly LC1. But the P condition must be false since the task is not present, so LC1 cannot be violated. We are guaranteed that there is a choice of non-conflicting minimal sets for the remaining tasks (by the WCF condition). We can assign the values corresponding to these minimal sets to those tasks and be assured that LC3 is satisfied. Since all variable corresponding to a particular task get assigned the same value, the external constraint is satisfied. We have a solution to the DyDisCSP.$\square$

Theorem 12 *Given a WCF Resource Allocation Problem* $\langle \mathcal{A}g, \Omega, \Theta \rangle$, $\Theta_{current} \subseteq \Theta$ *and the DyDisCSP obtained from Mapping II, if an assignment of values to variables in the DyDisCSP is a solution, then all tasks in* $\Theta_{current}$ *are performed.*

Proof: Let a solution to the DyDisCSP be given. We want to show that all tasks in $\Theta_{current}$ are performed. We proceed by contradiction. Let $T_r \in \Theta_{current}$ be a task that is not performed in the given solution state. Condition (i) of the **Notification Assumption** says some operation O_p^i, required by T_r will be executed and (by definition) succeed. LC1 requires the corresponding agent A_i, to assign a minimal set to some task which requires O_p^i. There may be many choices of tasks that require O_p^i. Suppose A_i chooses a task T_s. A_i assigns a minimal set, say t_s, to the variable $T_{s,i}$. The EC constraint will then require that all other agents A_j, who have a local copy of T_s called $T_{s,j}$, to assign $T_{s,j} = t_s$. In addition, if A_j has an operation O_q^j in the minimal set t_s, it will execute that operation. Also, we know that A_j is not already doing some other operation since t_s cannot conflict with any other chosen minimal set (by LC3).

We now have two cases. In case 1, suppose $T_s \neq T_r$. Condition (ii) of the **Notification Assumption** states that T_r is the only task that both requires O_p^i and is actually present. Thus, T_s cannot be present. By definition, if T_s is not present, it cannot be performed. If it cannot be performed, there cannot exist a minimal set of T_s where all operations succeed (def of "performed"). Therefore, some operation in t_s must fail. Let O_q^j be an operation of agent A_j that fails. Since A_j has assigned value $T_{s,j} = t_s$, LC2 is violated by A_j. This contradicts the fact we are in a solution state. Case 1 is not possible. This leaves case 2 where $T_s = T_r$. Then, all operations in t_s succeed and T_r is performed. We assumed T_r was not performed, so by contradiction, all tasks in $\Theta_{current}$ must be performed. $\square$

5.8 Mapping OC Problems into DCOP

In the previous sections we were able to represent dynamic problems only by limiting ourselves to SCF and WCF problems. In this section we consider OC problems where we must deal with an optimization problem. Since effective distributed constraint optimization algorithms for dynamic problems do not currently exist, we are forced to limit ourselves to static problems.

We assume we are given a *weight function* w: $\mathcal{T} \rightarrow N$ that quantifies the cost of not completing a task. The goal is for the agents to find a $\mathcal{T}_{ignore} \subseteq \mathcal{T}$ such that $\sum_{T \in \mathcal{T}_{ignore}} w(T)$ is minimized and there are enough agents to complete all tasks in $\mathcal{T} \setminus \mathcal{T}_{ignore}$.

We informally outline the mapping of OC problems into DCOP. While we do not formally prove its correctness, this mapping is presented mainly to illustrate the versatility of the constraint representation in its ability to model optimization-based distributed resource allocation problems. Mapping III converts an overconstrained distributed resource allocation problem into a DCOP thereby allowing an algorithm such as Adopt to be used to solve such problems.

Mapping III: Given a static Resource Allocation Problem $\langle \mathcal{A}g, \Omega, \Theta \rangle$, the corresponding DCOP is defined over a set of n variables.

- $A = \{A_1, A_2, ..., A_n\}$, one variable for each $A_i \in$ Ag.

The domain of each variable is given by:

- $\forall A_i \in \mathcal{A}g$, Dom($A_i$) = set of operations A_i could possibly execute.

The above is similar to Mapping I except that $\{yes, no\}$ values are not needed. This is because we no longer need to deal with ambiguity since we have assumed a static problem. This also means that our constraints will be static. We define an n-ary constraint for each task. For each task T, we define an n-ary constraint over all the agents/variables who could possibly contribute doing T. The constraint is a *valued* constraint whose cost function is shown in Figure 5.4. Suppose we have task T1 from Figure 5.2. If the maximum number of agents (all four agents A1,A2,A3,A4) choose to perform the operations for T1, then the agents pay zero cost on the n-ary constraint that

Minimal Set of T1 = {A1, A2, A3, A4}

N–ary constraint for T1:

A1	A2	A3	A4	Cost
T1	T1	T1	T1	0
T1	T1	T1	T2	w(T1)/4
T1	T1	T2	T2	w(T1)/2
T1	T3	T2	T2	w(T1)

Figure 5.4: N-ary constraint for Mapping III

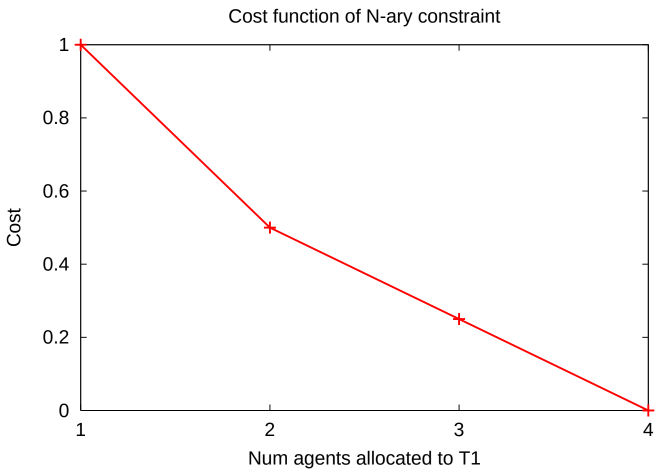


Figure 5.5: Graph of cost function for n-ary constraint for Mapping III

corresponds to T1. This is shown in first row of the cost table in Figure 5.4. If one of the agents decides not to allocate itself to T1 and instead do some other task, the cost on this constraint is increased to 1/4 the weight of the task. This is shown in the second row of the cost table. Similar for the third row. Finally, in the fourth row, only one agent allocated to the task. This is considered worthless and the agents pay the full cost for not performing this task. Figure 5.5 is a graphical depiction of this cost function.

One potential problem with Mapping III is that it requires agents to solve DCR problems containing n-ary constraints which can be expensive. However, similar to the technique used to change from Mapping I and II, we can convert Mapping III into its dual representation which has only binary constraints.

Chapter 6

Related Work

6.1 Related Work in Distributed Constraint Reasoning

This section discusses related work in distributed constraint reasoning for multiagent domains. Section 6.1.1 provides an discussion of work on distributed constraint satisfaction relevant to DCOP, while section 6.1.2 provides an overview of various existing approaches to DCOP.

6.1.1 Distributed Constraint Satisfaction

Yokoo, Hirayama and others have studied the DisCSP problem in depth and a family of sound and complete algorithms for solving these types of problems in a decentralized manner exist [50]. This has been an important advance and provides key insights that influence the work presented here. However, existing distributed search methods for DisCSP do not generalize easily to DCOP.

Armstrong and Durfee [1] investigate the effect of agent priority orderings on efficiency in DisCSP. They show that variable ordering heuristics from CSP can be reused as priority orderings in DisCSP and that dynamic reordering is also a useful technique. These results could potentially be generalized and applied to DCOP. Silaghi, Sam-Haroud and Faltings [40] present an alternative representation of DisCSP in which constraints are assigned to agents while variables are shared between agents. This approach allows the distributed constraint paradigm to be applied in distributed domains where constraints cannot be shared, perhaps for privacy reasons, but variables may be assigned to multiple agents. Representing DCOP in this manner is an interesting direction of future work.

6.1.2 Distributed Constraint Optimization

Table 6.1 outlines the state of the art in existing approaches to DCOP. Methods are parameterized by communication model (asynchronous or synchronous), completeness (guaranteed optimal solutions for DCOP), and "distributedness". We assume that a method is not distributed if all agents are required to communicate directly with a single agent irrespective of the underlying constraint network. The individual approaches are discussed further below.

Table 6.1: Characteristics of Distributed Constraint Optimization Methods

Method	Asynch?	Optimal?	Dist?
Satisfaction-Based Search[24][17]	N	N	Y
Local [16][12]	Y	N	Y
Synchronous Search [16]	N	Y	Y
Greedy Repair [22]	N	N	N
Asynchronous Best-First Search (Adopt)	Y	Y	Y

Satisfaction-based methods. This method leverages existing DisCSP search algorithms to solve special classes of DCOP, e.g. overconstrained DisCSP. In overconstrained DisCSP, the goal is to optimize a global objective function by relaxing constraints since no completely satisfactory solution may be possible. The approach typically relies on converting the DCOP into a sequence of satisfaction problems in order to allow the use of a DisCSP algorithm. This can be done by iteratively removing constraints from the problem until a satisfactory solution is found. However, a drawback of this approach is that agents need to repeatedly synchronize to remove constraints (although the satisfaction-based search component may be asynchronous). Hirayama and Yokoo [17] show that this approach can find optimal solutions for a limited subclass of optimization problems, namely overconstrained DisCSP in which solutions can be structured into hierarchical classes. Liu and Sycara [24] present another similar iterative relaxation method, Anchor&Ascend, for heuristic search in a job-shop scheduling problem. These satisfaction-based methods fail to generalize to the DCOP defined in this work since agents are not able to asynchronously determine which constraints should be relaxed to obtain the optimal solution.

Local Methods. In this approach, agents are oblivious to non-local costs and simply attempt to minimize costs with respect to neighboring agents. Methods such as random value change or dynamic priority ordering may be used for escaping local minima. In this method, no guarantees on solution quality are available even if given unlimited execution time. Furthermore, agents cannot know the quality of the solution they have obtained. Examples of this approach include the Iterative Distributed Breakout (IDB) algorithm[16]. This algorithm utilizes the Satifaction-Based approach described above, and so is limited in the type of DCOP it can address. In particular, IDB is applicable to a particular class of DCOP in which agents wish to minimize the maximum cost incurred at any agent. This type of criterion function has the special property that some agent can always locally determine the global cost of the current solution without knowledge of the cost incurred at other agents. For this class of DCOP, IDB is empirically shown to find good solutions quickly but cannot guarantee optimality.

Fitzpatrick and Meertens [12] present a simple distributed stochastic algorithm for minimizing the number of conflicts in an overconstrained graph coloring problem. Agents change variable value with some fixed probability in order to avoid concurrent moves. No method for escaping local minimum is used. The algorithm is shown empirically to quickly reduce the number of conflicts in large sparse graphs, even in the face of noisy/lossy communication. It is unknown how this approach would work in general since the quality of local minima can be arbitrarily poor.

Synchronous Search. This approach can be characterized as simulating a centralized search method in a distributed environment by imposing synchronous, sequential execution on the agents. It is seemingly straightforward to simulate centralized search algorithms in this manner. An example includes SynchBB (Synchronous Branch and Bound) [16]. While this approach yields an optimal distributed algorithm, the imposition of synchronous, sequential execution can be a significant drawback.

Greedy Repair. Lemaitre and Verfaille [22] describe an incomplete method for solving general constraint optimization problems. They address the problem of distributed variables by requiring a leader agent to collect global cost information. Agents then perform a greedy repair search where only one agent is allowed to change variable value at a time. Since all agents must communicate with a single leader agent, the approach may not apply in situations where agents may only communicate with neighboring agents.

6.1.3 Other Work in DCOP

R. Dechter, A. Dechter, and Pearl [9] present a theoretical analysis of the constraint optimization problem establishing complexity results in terms of the structure of the constraint graph and global optimization function. In addition, they outline an approach for distributed search for the optimal solution based on dynamic programming, although no algorithm or empirical results are given. They do not deal with asynchronous changes to global state or timeliness of solution.

Parunak *et al* [32] describe the application of distributed constraint optimization to the design of systems that require interdependent sub-components to be assembled in a manufacturing domain. The domain illustrates the unique difficulties of interdependencies between sub-problems in distributed problem solving and illustrates the applicability of the distributed constraint representation. Frei and Faltings [13] focus on modelling bandwidth resource allocation as a CSP. Although they do not deal with distributed systems, they show how the use of abstraction techniques in the constraint modelling of real problems results in tractable formulations.

6.2 Related Work in Multiagent Systems

A variety of researchers have focused on formalizing resource allocation as a centralized CSP[13]. In addition, the Dynamic Constraint Satisfaction Problem has been studied in the centralized case by [38]. In centralized CSP, there is no distribution or ambiguity during the problem solving process. However, the fact that the resource allocation problem is inherently distributed in many domains means that ambiguity must be dealt with. We also categorize different resource allocation problems and provide detailed complexity results.

Distributed sensor networks have gained significant attention in recent years. Sensor hardware nodes are becoming more sophisticated and able to support increasing levels of both computation and communication. This trend, in turn, has lead to the consideration of the Distributed AI (DAI) and the Multiagent perspective for addressing technical problems in distributed sensor networks [28] [27] [18] [36] [41] [45]. Mailler and Lesser [28] propose the SPAM protocol for multi-stage mediated negotiation. In multi-stage negotiation, a quick solution is found in the first stage and if remaining time is available, a second stage attempts to refine the solution to improve global quality. The negotiation is mediated by multiple managers who negotiate amongst themselves on behalf of the sensor agents. SPAM has also been generalized to more abstract DisCSP domains [27]. Soh and Tsatsoulis [41] describe a case-based reasoning approach where agents choose different negotiation strategies depending on environmental circumstances in order to collaboratively track moving targets. Vincent et al. [45] and Horling et al. [18] bring existing multi-agent technologies, such as the TAEMS modelling language [10] and the Design-to-Criteria plan scheduler [46], to bear on the distributed sensor network problem. These approaches report positive results in customizing and applying existing DAI technologies to the specific problem of coordination in distributed sensor networks.

There is significant research in the area of distributed resource allocation. For instance, Liu and Sycara [25] address resource allocation in the distributed job-shop scheduling problem. The solution technique presented is an extension of local dispatch scheduling – the extension allows agents to use non-local information to make their local decisions. Schedules are shown to improve using this technique. Chia et al's [7] work on job-shop scheduling for airport ground service schedules is another example of distributed resource allocation. The authors are able to reduce schedule inefficiencies by allowing agents to communicate heuristic domain specific information about their local jobs. Finally, the Distributed Vehicle Monitoring Testbed (DVMT) of Lesser et al. [23] is well known in distributed AI as a domain for distributed problem solving. A set of problem-solving nodes must cooperate and integrate distributed sensing data to accurately track a moving vehicle. This domain is inherently distributed and exhibits both dynamics and ambiguity. To summarize, while previous work in distributed resource allocation has been effective for particular problem domains, a formalization of the general problem which allows tractable subclasses to be identified, is yet to be developed.

A recent approach to distributed resource allocation that has received significant attention is that of market-based systems, or multi-agent auctions[49]. We view these approaches as complementary to a distributed constraints approach. Indeed, Sandholm and Suri [35] discuss the need for non-price attributes and explicit constraints in conjuction with market protocols. Briefly, price-based search techniques coordinate a set of agents by allowing them to buy and sell goods in order to maximize local utility. This approach has been used effectively to structure distributed search in many multi-agent domains including multi-commodity flows [49], multi-agent task allocation [47] and even distributed propositional satisfiability [48].

One of the main features of market-based systems is that agents communicate only in terms of prices. The price of a good is a compact way to implicitly convey to an agent non-local information about the global utility of obtaining some good. This is a key advantage in open systems where agents cannot be trusted. However, the market-based approach may be unnecessarily restrictive in many collaborative multiagent domains. In collaborative domains, agents may be able to reach global optimal solutions faster by exchanging more information beyond prices. However, the market-based approach in collaborative multi-robot scenarios has been investigated by Gerkey and Mataric [14]. In conclusion, much research remains to be done to compare and contrast market-based systems and distributed constraints as competing technologies for distributed resource allocation.

Chapter 7

Conclusion

Motivated by the need to design collaborative agents that are able to reason about how their decisions interact with each other and with a global objective, this dissertation makes several contributions to the field of Multiagent Systems. We review these contributions next.

- We have presented the Adopt algorithm for Distributed Constraint Optimization (Chapter 3). In Adopt, agents execute in a completely decentralized manner and communicate asynchronously and locally. The algorithm is proven to terminate with the globally optimal solution. Empirical results in a benchmark domain show that Adopt achieves significant orders of magnitude efficiency gains over competing methods. Additionally, we show that the algorithm is robust to loss of messages.

The above benefits are derived from our general idea that in a distributed environment agents should perform optimization based on conservative solution quality estimates. By communicating conservative estimates asynchronously, we allow agents to make the best decisions possible given currently available information. If more accurate estimates are asynchronously received from others, an agent can revise its decisions as necessary.

- We have proposed bounded-error approximation as a flexible method for dealing with domains where time for problem solving is limited (Chapter 4). The key idea is to allow the user (or potentially an agent itself) to provide the algorithm with an allowance on suboptimality (an error bound). We show that by increasing the given error bound, the time to solution is decreased significantly. These results are significant because, in contrast to incomplete search methods, Adopt provides the ability to find solutions faster when time is limited but without giving up theoretical guarantees on solution quality.

- We have given a detailed complexity analysis of distributed resource allocation and provided general mapping strategies for representing it via distributed constraints (Chapter 5). The mapping strategies are theoretically proven to correctly represent distributed resource allocation problems. This contribution is significant because it enables existing distributed constraint technologies to be brought to bear directly onto the distributed resource allocation problem without significant re-modeling effort.

Chapter 8

Future Work

- Dynamic, real-time environments. A key challenge for distributed inter-agent reasoning is operating in rapidly changing environments. Existing methods assume a "one-shot" problem solving process, where agents find a solution and terminate. However, many coordination problems require agents to dynamically modify their decisions over time as environmental changes occur. How can this be done efficiently, without restarting problem solving from scratch?

  The distributed reasoning techniques presented in this dissertation provide significant steps towards addressing this challenge. Indeed, dynamic environments only increase the saliency of our key premise: that obtaining global knowledge in a distributed environment is difficult. In such situations, it is essential that agents are able to make the best decisions possible with currently available information. If things change, the information may be updated and new decisions made. In addition, our techniques that allow agents to find approximate solutions quickly, but improve those solutions if time permits, provides a promising path forward for flexibly dealing with real-time situations.

- Loss of agents. Agent failure is an important problem in many real-world domains. Sensor nodes may cease to function, robots may run out of battery power, or adversaries may interfere with agents in the system. Designing algorithms that are robust to such failures is an open question in Distributed Constraint Reasoning.

  One possible approach in Adopt is to devise a method to dynamically update the agent ordering when some agent is lost. For example, the children of the lost agent may reconnect to the parent of the lost agent. It may be possible to seemlessly continue algorithm execution by patching the tree structure in this manner. The difficulty lies in detecting that an agent has been lost, identifing how to patch the tree in a distributed manner, and ensuring that the algorithm's optimality guarantee is maintained.

- Rogue agents. In open domains, such as multiagent systems that operate over the internet, not all agents can be trusted. Agents may send messages to others that contain false information in order to further private goals. Multiagent research in incentive-compatibility is concerned with ensuring that agents are truthful. In non-collaborative domains, if such a method can be used to ensure that agents communicate truthfully, then Adopt may be applied. Thus, while Adopt was designed with collaborative agents in mind, it is rather indifferent to whether agents are actually collaborative or not, provided they are compeled to tell the truth through some other mechanism.

- Scale-up. How can we perform distributed optimization with quality guarantees as we increase the number of agents, say to 100 or 1000s? At this scale, decomposition becomes a key technique for dealing with problems. Agents may be able to decompose a large DCOP into smaller subproblems corresponding to smaller communities of agents. A key outstanding challenge is how to do the decomposition so that the small communities can do problem solving independently without large or unknown effects on overall quality.

Reference List

[1] A. Armstrong and E. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proc of International Joint Conference on Artificial Intelligence*, 1997.

[2] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving tcp performance over wireless links. In *Proceedings of ACM SIGCOMM*, 1996.

[3] A. Barrett. Autonomy architectures for a constellation of spacecraft. In *International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS)*, 1999.

[4] R. Caulder, J.E. Smith, A.J. Courtemanche, M.F. Mar, and A.Z. Ceranowicz. Modsaf behavior simulation and control. In *Proceedings of the Conference on Computer Generated Forces and Behavioral Representation*, 1993.

[5] H. Chalupsky, Y. Gil, C.A. Knoblock, K. Lerman, J. Oh, D.V. Pynadath, T.A. Russ, and M. Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of Innovative Applications of Artificial Intelligence Conference*, 2001.

[6] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 1985.

[7] M. Chia, D. Neiman, and V. Lesser. Poaching and distraction in asynchronous agent activities. In *International Conference on Multiagent Systems*, 1998.

[8] S.E Conry, K. Kuwabara, V.A. Lesser, and R.A. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Trans. Systems, Man and Cybernetics*, 1991.

[9] R. Dechter, A. Dechter, and J. Pearl. Optimization in constraint networks. In *Influence Diagrams, Belief Nets, and Decision Analysis*. 1990.

[10] K. Decker and V. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 2(4), 1993.

[11] C. Fernandez, R. Bejar, B. Krishnamachari, and C. Gomes. Communication and computation in distributed csp algorithms. In *Principles and Practice of Constraint Programming*, 2002.

[12] S. Fitzpatrick and L. Meertens. An experimental assessment of a stochastic, anytime, decentralized, soft colourer for sparse graphs. In *Stochastic Algorithms: Foundations and Applications, Proceedings SAGA*, 2001.

[13] C. Frei and B. Faltings. Resource allocation in networks using abstraction and constraint satisfaction techniques. In *Proc of Constraint Programming*, 1999.

[14] Brian P. Gerkey and Maja J Mataric. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation, Special Issue on Multi-robot Systems*, pages 758–768, 2002.

[15] Eric Hansen and Shlomo Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, pages 139–157, 2001.

[16] K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In G. Smolka, editor, *Principles and Practice of Constraint Programming*, pages 222–236. 1997.

[17] K. Hirayama and M. Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of International Conference on Multiagent Systems*, 2000.

[18] B. Horling, R. Vincent, R. Mailler, J. Shen, R. Becker, K. Rawlins, and V. Lesser. Distributed sensor network for real time tracking. In *Proceedings of the 5th International Conference on Autonomous Agents*, 2001.

[19] H. Jung, M. Tambe, A. Barrett, and B. Clement. Enabling efficient conflict resolution in multiple spacecraft missions via dcsp. In *Proceedings of the NASA workshop on planning and scheduling*, 2002.

[20] H. Kitano, S. Todokoro, I. Noda, H. Matsubara, and T Takahashi. Robocup rescue: Search and rescue in large-scale disaster as a domain for autonomous agents research. In *Proceedings of the IEEE International Conference on System, Man, and Cybernetics*, 1999.

[21] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[22] M. Lemaitre and G. Verfaillie. An incomplete method for solving distributed valued constraint satisfaction problems. In *Proceedings of the AAAI Workshop on Constraints and Agents*, 1997.

[23] V. Lesser and D. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3), 1983.

[24] J. Liu and K. Sycara. Exploiting problem structure for distributed constraint optimization. In *Proceedings of International Conference on Multi-Agent Systems*, 1995.

[25] J. Liu and K. Sycara. Multiagent coordination in tightly coupled task scheduling. In *Proceedings of International Conference on Multi-Agent Systems*, 1996.

[26] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[27] R. Mailler and V. Lesser. A mediation based protocol for distributed constraint satisfaction. In *The Fourth International Workshop on Distributed Constraint Reasoning*, 2003.

[28] R. Mailler, V. Lesser, and B. Horling. Cooperative negotiation for soft real-time distributed resource allocation. In *Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, 2003.

[29] P. J. Modi, H. Jung, W. Shen, M. Tambe, and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. In *Principles and Practice of Constraint Programming*, 2001.

[30] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe. Modeling tcp throughput: A simple model and its empirical validation. In *ACM Computer Communications Review: Proceedings of SIGCOMM 1998*, 1998.

[31] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[32] V. Parunak, A. Ward, M. Fleischer, J. Sauter, and T. Chang. Distributed component-centered design as agent-based distributed constraint optimization. In *Proc. of the AAAI Workshop on Constraints and Agents*, 1997.

[33] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[34] Zsofia Ruttkay. Constraint satisfaction - a survey. *CWI Quarterly*, 11, 1998.

[35] T. Sandholm and Subhash Suri. Side constraints and non-price attributes in markets. In *International Joint Conference on Artificial Intelligence Workshop on Distributed Constraint Reasoning*, 2001.

[36] P. Scerri, P.J. Modi, W. Shen, and M. Tambe. Are multiagent algorithms relevant for robotics applications? a case study of distributed constraint algorithms. In *ACM Symposium on Applied Computing*, 2003.

[37] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *International Joint Conference on Artificial Intelligence*, 1995.

[38] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 1994.

[39] W.-M. Shen and Mark Yim. Self-reconfigurable robots. *IEEE Transactions on Mechatronics*, 7(4), 2002.

[40] M.C. Silaghi, D. Sam-Haroud, and Boi Faltings. Asynchronous search with aggregations. In *Proceedings of National Conference on Artificial Intelligence*, 2000.

[41] L-K. Soh and C. Tsatsoulis. Reflective negotiating agents for real-time multisensor target tracking. In *International Joint Conference On Artificial Intelligence*, 2001.

[42] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1994.

[43] Katia Sycara, Steven F Roth, Norman Sadeh-Koniecpol, and Mark S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:1446–1461, 1991.

[44] BAE Systems. Ecm challenge problem, http://www.sanders.com/ants/ecm.htm. 2001.

[45] R. Vincent, B. Horling, V. Lesser, and T. Wagner. Implementing soft real-time agent control. In *Proceedings of the 5th International Conference on Autonomous Agents*, 2001.

[46] T. Wagner and V. Lesser. Criteria-directed heuristic task scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 1998.

[47] W. Walsh and M. Wellman. A market protocol for decentralized task allocation. In *International Conference on Multi-Agent Systems*, 1998.

[48] W. Walsh, M. Yokoo, K. Hirayama, and M. Wellman. On market-inspired approaches to propositional satisfiability. In *International Joint Conference on Artificial Intelligence*, 2001.

[49] Michael Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, pages 1–23, 1993.

[50] M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.

[51] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *Proceedings of International Conference on Multi-Agent Systems*, 1996.

[52] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of International Conference on Multi-agent Systems*, 1998.

[53] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2002.